Durham E-Theses

Reasoning about Locks and Transactions in Concurrent Programs

BARNETT, GRANVILLE

How to cite:

BARNETT, GRANVILLE (2014) *Reasoning about Locks and Transactions in Concurrent Programs*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/10573/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, Durham University, University Office, Old Elvet, Durham DH1 3HP
e-mail: e-theses.admin@dur.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

Reasoning about Locks and Transactions in Concurrent Programs

Granville Barnett

School of Engineering and Computing Sciences

Durham University

A thesis submitted for the degree of

Doctor of Philosophy

2013

Contents

List of Figures

The research presented in this thesis is the original work of the author, unless stated otherwise. The copyright of this thesis rests with the author. No quotation from it should be published without the author's prior written consent and information derived from it should be acknowledged.

Acknowledgements

I thank my supervisor Professor Shengchao Qin for his tutilige and support over the years. My thanks also to Professor Iain Stewart and an EPSRC Doctoral Training Award from Durham University, without which this PhD would not have been possible. I also wish to thank: Ryuta Arisaka, Professor Paul Roe, Dr Wayne Kelly, Professor John Gough, Dr Keshav Dahal, Professor Peter Cowling, Professor Chin Wei Ngan, Professor Huibiao Zhu, Dr Steven Hand, Andrew Craik, Richard Mason, Darryl Cain, Joao Ferreira, Guanhua He, Chenguang Luo, Le Duy Khanh and Mengda He.

Wariya – thank you for your love, support and patience.

Mum – thank you for always believing in me.

Abstract

The aim of this thesis is to present novel techniques for reasoning about the *dynamic* and *static* semantics of concurrent programs that use locks and transactions to isolate accesses to shared memory. We use *moverness* to characterise the observational semantics of reads issued by locks and transactions under the simpler semantics of *free*, *left*, *right* and *both movers*. The second contribution is *guaranteed transactions* which are a safer alternative to locks and the privatisation/publication idioms for specific scenarios. Guaranteed transactions facilitate a simpler pessimistic coordination semantics than locks, but offer most of the conveniences that have made transactions appealing. Finally, we present a static analysis for reasoning about the isolation of a program that uses locks and transactions. If our isolation algorithm determines that all the accesses issued by a program are isolated, then the program is declared *data-race-free*.

Chapter 1

Introduction

1.1 Background

1.1.1 Chip Multiprocessors

Failure to economically address heat dissipation in uniprocessors has resulted in industry adoption of *chip multi-processors* (CMP) [Olukotun et al., 1996]. Each CMP comprises a number of homogeneous *processing elements* (PE). By contrast to the PE found in a uniprocessor, the PEs in CMPs consume less power and dissipate less heat. Desktop PCs, laptops and most recent tablets and smart phones comprise CMPs. The transition to CMPs has a large impact on software. Designing software for the uniprocessor was relatively simple: solutions were described as a sequence of linear commands, and every other year or so the solution would receive a significant speedup [Schaller, 1997]. This sort of design under present-day hardware gains little to no speedup [Sutter and Larus, 2005]. Exploiting CMPs requires a fundamental shift in software design: instead

of focusing on linear execution (*vertical scaling*), we now focus our efforts on partitioning work into tasks which can be distributed across the PEs of a CMP (*horizontal scaling*). Figures 1.1 and 1.2 show vertical and respectively horizontal scaling under CMPs. The goal of horizontal scaling is relatively simple: we would like to design software in such a way that it can take advantage of all the PEs of a CMP, irrespective of whether the CMP comprises four or four hundred PEs. Software designs that embrace horizontal scaling can expect favourable speedups as CMPs with larger quantities of PEs are released. For example, an algorithm that scales horizontally can potentially run twice as fast on a CMP with four PEs than it did on a CMP with two PEs, and so on. Linear speedups such as the previous example are the gold standard for software targeting CMPs. In theory CMPs are spawning an exciting era in computing: problems that were previously the domain of supercomputing are now computationally tractable on consumer grade hardware. However, as will shortly be illustrated, the correct design of such programs using the current tools is steeped in technical idiosyncrasies, making the task of exploiting CMPs in practice a difficult and error-prone task.

1.1.2 Threads

Horizontal scaling requires the use of *threads* [Butenhof, 1997] . Before the importance of threads can be understood we need to describe their role in modern operating systems. Let us assume we have a valid C program defined in the file `program.c` which has the single method `main`. At the moment `program.c` is just a text file. To create something the machine can understand we need to compile and link `program.c` using the command `CC program.c`, where `CC` is a C compiler.

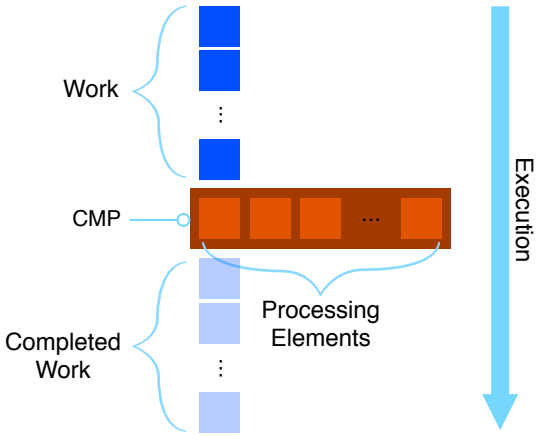


Figure 1.1: A vertically scaled program describes its computation as a linear sequence of commands. This linear sequence can only utilise a single PE, irrespective of whether the other PEs of the CMP are being utilised.

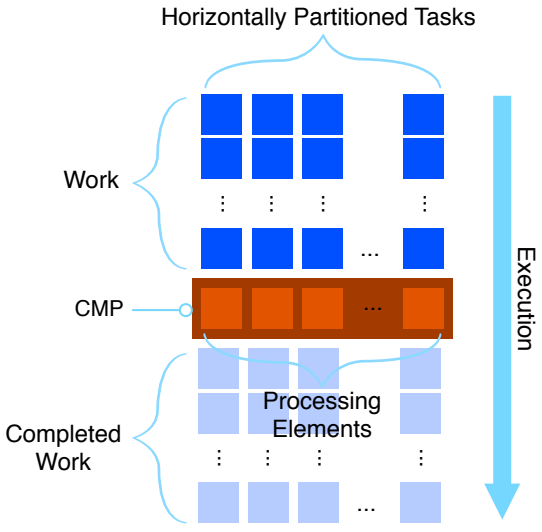


Figure 1.2: A horizontally scaled program describes its computation as a series of partitioned tasks. A task is defined by a linear sequence of commands. Tasks can be executed by the available PEs of the CMP.

The result of the previous step is the *binary image* `a.out`. We do not need to know the detailed contents of `a.out`, just that it contains the machine instructions that model the high-level commands defined in `program.c`. To *execute* our program we issue a command such as `./a.out` from a UNIX terminal. When we issue this command the operating system performs a number of steps: creation of a new *process*; assigning *virtual memory* to the newly created process; loading the binary image `a.out` into the process's memory; and creation of a *main thread*, so-called because it executes the user defined method labelled `main`. The `main` method is often known as being the *entry point* due to it being the earliest point where user defined commands are executed. Each thread entails a stack and possibly some private memory known as *thread local storage*. The thread's stack facilitates method calls. A process has at least one thread, otherwise it can perform no meaningful work.

1.1.3 Tasks

Each PE of a CMP can execute one thread at a time. The PEs of some CMPs, such as those manufactured by Intel with Hyper-Threading [Intel, 2013a], can execute two threads at a time. Utilising the PEs of a CMP requires a program to partition its work into *tasks*. A task is described as a method and can be passed to a thread to execute. A process that creates multiple threads during its lifetime is said to be *multi-threaded*. The process that models the execution of `program.c` is not multi-threaded as it comprises only the main thread. That is, the process will only utilise one PE, even if several PEs of the CMP are available, just like Figure 1.1. To better utilise a CMP our process needs to create additional threads

and map tasks to those threads. The tasks delegated to these additional threads may execute *concurrently* like in Figure 1.2. That is, each PE of the CMP may execute a distinct thread of the process at the same time.

1.1.4 Thread Scheduling

Typically more threads than PEs exist. The job of an operating system's *thread scheduler* is to map threads to PEs. There are two types of scheduling approaches: *non-preemptive* and *preemptive*. Under non-preemptive scheduling the threads of a process utilise the CMP for as long as they need to execute; however, a thread can *voluntarily yield* control of a PE if it wishes, e.g. it may yield while waiting for some I/O to complete. Non-preemptive scheduling is a simple model of cooperative computing but an unfair one. For example, a thread may infrequently or never yield, starving other threads from utilising the CMP. In response, most modern operating systems, including Linux, OSX and Windows, use preemptive scheduling. A preemptive scheduler generally uses *time quantums* and domain-specific heuristics to ensure that the PEs of a CMP are *fairly* shared between the threads of processes. Under preemptive scheduling each thread is given a time quantum, the maximum amount of contiguous time it may utilise a PE, and a priority. A thread implicitly yields if it terminates within its allotted time quantum. A preemptive scheduler is free at any time during a thread's utilisation of a PE to *context switch* it out in favour of a waiting thread. A context switch generally entails: (1) saving the state of the thread currently utilising the PE; (2) placing that thread in the waiting queue; and (3) mapping a thread from the waiting queue to the now vacant PE. The heuristics used to select the next thread

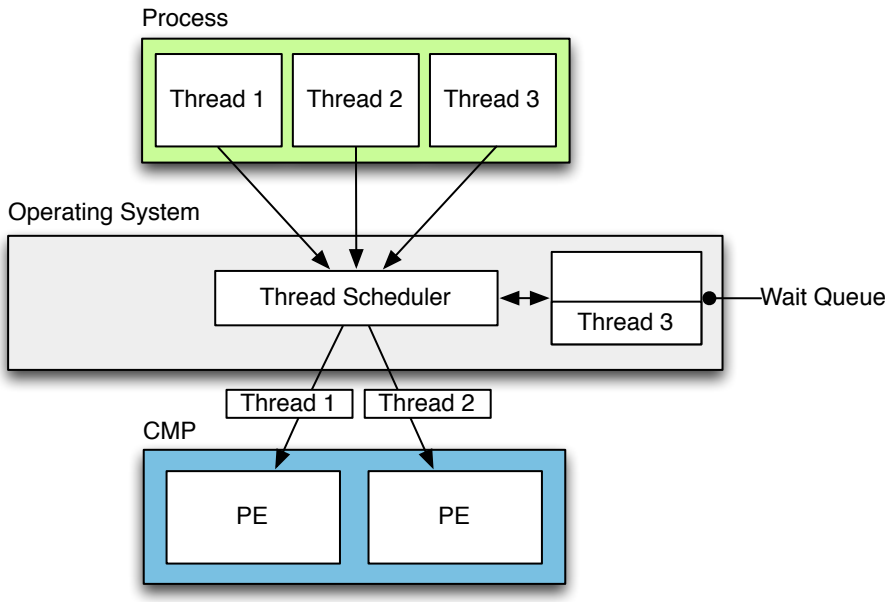


Figure 1.3: Three threads contend utilisation of the CMP's two PEs. Threads 1 and 2 are scheduled to utilise the CMP by the operating system's thread scheduler; Thread 3 is placed in the wait queue.

to run and the technical details of context switching are irrelevant. However, the fact that a thread can be usurped from utilising a PE at any time is very important. Figure 1.3 describes a scheduling scenario with three threads from the same process contending utilisation of a CMP with two PEs.

1.1.5 Accessing Shared Memory

The threads created during the lifetime of a process share the process's virtual memory. We will refer to this memory as *shared memory*. Executing a thread's task results in the thread issuing a sequence of low-level instructions. These instructions are taken from the binary image `a.out`. For example, a thread that increments the integer value of a variable `x` by one, described by the high-level

command `x := x + 1`, is modelled by a sequence of low-level instructions, such as the pseudo-instructions `load x; push_int 1; add; store x`. Figure 1.4 shows the operation of these instructions. There are two important concepts on display here: (1) a high-level command is implemented as a sequence of instructions; and (2) these instructions may issue *accesses* (*reads* and *writes*) to a process's shared memory, e.g. `load x` reads `x` and `store x` writes `x`. The low-level representation of a high-level program's commands, in conjunction with the operating system's preemptive scheduling, can result in a number of program defects exclusive to multi-threaded programs.

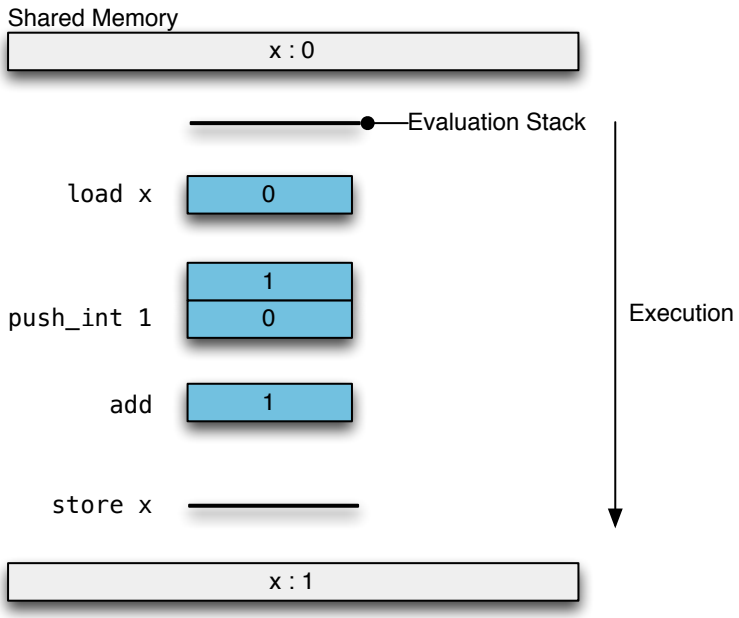


Figure 1.4: Incrementing `x`'s value: `load x` pushes `x`'s current value onto the evaluation stack; `push_int 1` pushes the integer literal 1; `add` pops the two values on the stack and pushes the result of its addition; `store` pops the value off the stack and stores it in `x`.

Figure 1.5 (a) gives a program where two threads increment the value of the *shared variable* `x`. We say a variable is shared if it resides in a process's shared

```
Int x;
x := 0;
```

Thread 1	Thread 2
`x := x + 1`	`x := x + 1`

(a)

```
Int x;
x := 0;
```

Thread 1	Thread 2

`load x`
`push_int 1`
`add`
`store x`
	`load x`
	`push_int 1`
	`add`
	`store x`

(b)

Figure 1.5: (a) Threads 1 and 2 increment the shared variable `x`. The double bars `||` denote the commands are executed concurrently. (b) Is the instruction representation of (a). Instructions are executed as described in Figure 1.4. Each thread has its own evaluation stack.

```
Int x;
x := 0;
```

Thread 1	Thread 2
`load x`	
	`load x`
	`push_int 1`
	`add`
`push_int 1`	
`add`	`store x`
`store x`	

Figure 1.6: Scheduling of Figure 1.5 (b) that leads to a data race on `x`. Thread 1 reads 0 as the value of `x`, then is preempted; Thread 2 reads 0 as the value of `x` and subsequently increments and writes 1 to `x` in shared memory; Thread 1 resumes execution and writes 1 to `x`.

memory. Recall that each thread of a process may access the data stored in its shared memory. Figure 1.5 (b) shows the low-level representation of Figure 1.5 (a). Each instruction takes place as an indivisible step: a preemptive scheduler cannot context switch a thread while it is executing an instruction; however, it can context switch a thread that is *between* executing instructions. Figure 1.6 shows a possible concurrent scheduling of Figure 1.5 (b). Here, the scheduler tries

to fairly share the uniprocessor's single PE between two threads. The initial value of `x` is 0, and each thread increments `x` by 1, so we expect to observe 2 for `x`'s final value. However, we observe 1. Our program has been subject to a *data race* [Unger, 1995]: the final value observed for `x` depends on the relative ordering of the instructions issued by each thread. The order that instructions are issued is dependent upon the operating system's thread scheduler. It is possible we could execute Figure 1.5 (a) several times on the same hardware and never observe 1 for `x`'s final value. If our process comprised more threads, each incrementing `x`, then the set of observable final values for `x` increases, and the schedules that reproduce the set of incorrect values of `x` grows. Data races are often hard to detect, e.g. in Figure 1.6 we observed 1 for the final value of `x`: logically this value is incorrect, despite 1 being an integer. Data races become harder to detect when advanced data types are used, e.g. user defined classes and data types which span multiple words in size. A programmer, suspecting the presence of a data race, may seek assistance from his language's compiler and debugger. A compiler for Java and C++ will provide no help. Success may be had with a debugger but only if he has an idea of where the data race originated. Let us suppose our programmer knows where to begin his search during a debugger session: he must still deal with the preemptive scheduling of the operating system; moreover, it is possible that use of the debugger affects access contention within the attached process due to the overhead of the debugger's instrumentation code.

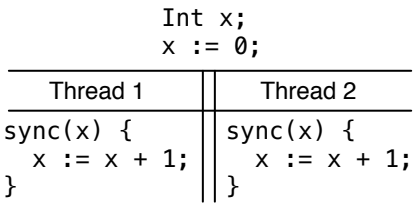


Figure 1.7: Using locks to remove the data race in Figure 1.5 (a).

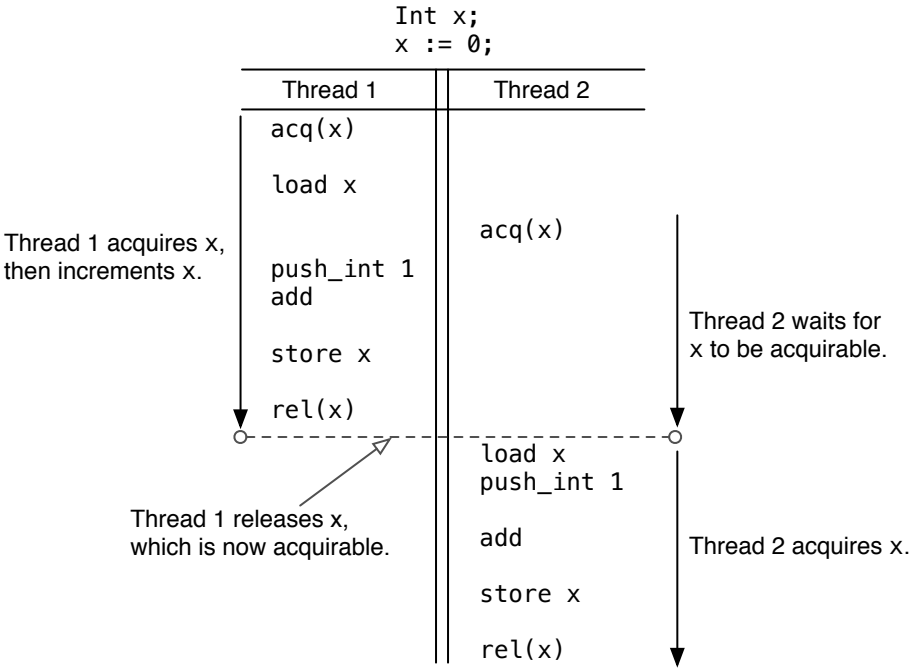


Figure 1.8: A scheduling of the instructions that represent Figure 1.7. Acquisition (`acq`) and release (`rel`) of `x` results in its increments being serialised. The final value observed for `x` is 2.

1.1.6 Coordination

Preventing data races requires the use of *coordination*. When employed correctly coordination facilitates thread exclusion.

1.1.6.1 Locks

Mutual exclusion is facilitated by a *binary semaphore* [Dijkstra, 1968]. A binary semaphore is known as a *mutex*. Let us use `sync(v) { c }` to mean that in order to execute the program commands `c` we must have *acquired* the mutex `v`; when `c` has completed executing `v` is *released*. A thread can acquire `v` if and only if another thread has not already acquired it; `v` becomes acquirable upon its release by the thread that currently has it acquired. Conceptually we can think of `v` as being released before any user defined program commands are run. That is, `v` is initially acquirable when the user's program text is executed. Figure 1.7 shows a version of Figure 1.5 (a) that uses the `sync` construct to remove the data race on `x`. We say that Figure 1.7 is *data-race-free* (DRF). Figure 1.8 shows how `sync` works at the instruction-level. We will refer to `sync(v) { c }` as a *lock* and permit any variable `v` to be used as a lock's mutex. The accesses issued by locks in distinct threads are *isolated* if and only if the locks use the same mutex. Figure 1.7 showed how easy it was to remove the data race on `x`; by contrast, Figures 1.9 and 1.10 show how simple it is to get locking wrong. Figure 1.9 (a) has a data race on `x` as thread 1 acquires `x` and then increments `x`; however, thread 2 issues its increment of `x` without having acquired `x`. Figure 1.10 (a) comprises a data race on `x` as each thread's lock uses a different mutex. Both Figures 1.9 (a) and 1.10 (a) are semantically equivalent to the accesses issued by Figure 1.5. A compiler will not warn the programmer of his failure to mutually exclude accesses to `x`, despite being obvious that was his intention.

The problem with locks is that in most languages they are a library facility [Butenhof, 1997; Oaks and Wong, 2004]. That is, a programming language does

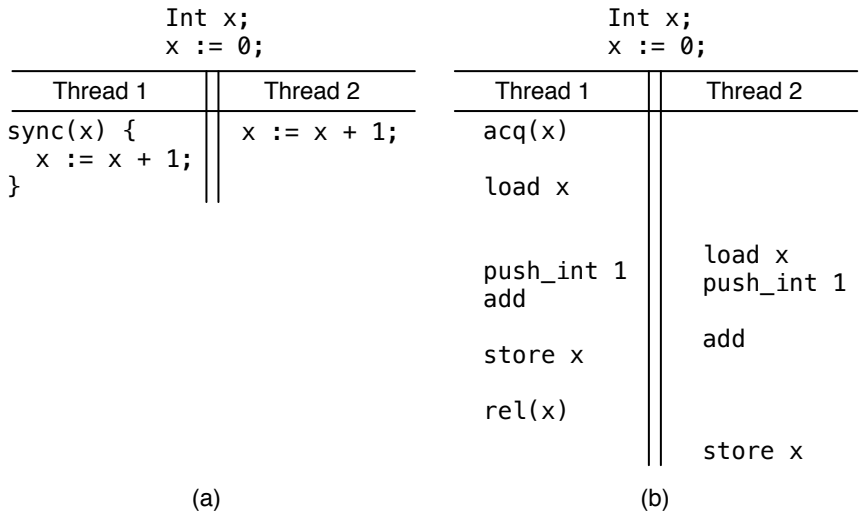


Figure 1.9: The increments of `x` are not isolated. Thread 1 issues its write of `x` while protected on `x`; thread 2 writes `x` irrespectively.

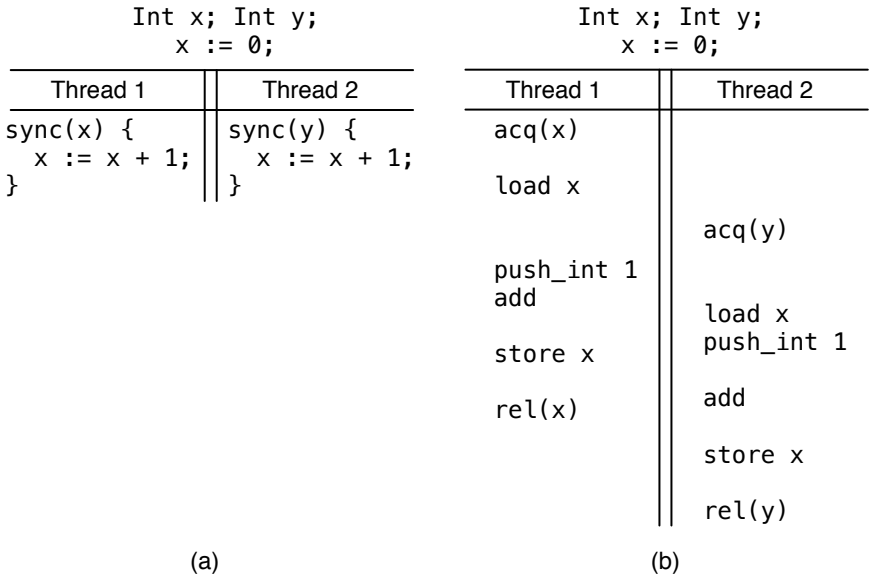


Figure 1.10: The increments of `x` are not serialised as each thread uses a different mutex to isolate its write of `x`.

not semantically treat an access issued within a lock any different to one issued outside of a lock. A programmer who works on a codebase that uses locks often

relies on program comments to determine what lock or locks should be acquired before accessing a particular bit of shared memory. These comments also often describe the *order* mutexes are to be acquired in. Acquisition and release orders are very important for mutexes. Figure 1.11 (a) shows a program where each thread acquires the mutexes `x` and `y` in opposing orders. Here, the opposing acquisition orders results in a program defect known as *deadlock* [Zöbel, 1983]. For example, consider the scheduling of acquire/release's given in Figure 1.11 (b) for the program in Figure 1.11 (a). Thread 1 acquires `x` then thread 2 acquires `y`. Neither thread can make any subsequent progress until the other thread releases their respective mutex. Unfortunately, neither thread can release their mutex until the other thread releases theirs. Both thread's will never make any further progress. Deadlock can be considered a simpler defect to diagnose than a data race. For example, in a debugger session we can observe that threads 1 and 2 are making no progress.

1.1.6.2 Software Transactional Memory

Software transactional memory (STM) [Shavit and Touitou, 1995] is another form of coordination. Under STM we issue accesses to shared memory using a *transaction*. A transaction in STM is similar to a transaction under a relational database management system (RDBMS). A transaction under a RDBMS exhibits the following properties: *Atomicity* – the effect of a transaction appears to take effect as a single step or not at all; *Consistancy* – only committed transactions contribute their effect to the underlying store; *Isolation* – transactional accesses are isolated with respect to other transactional accesses; and *Durability* – the underlying store persists, irrespective of whether the program executing the transaction

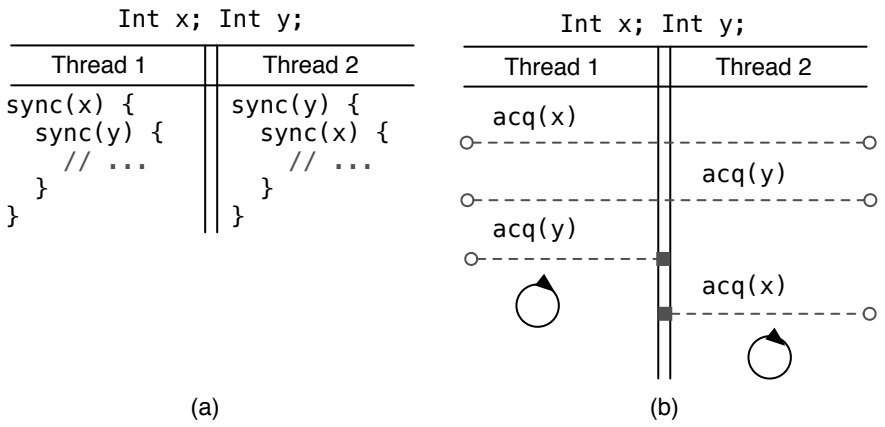


Figure 1.11: (a) The locks of threads 1 and 2 acquire `x` and `y` in reverse orders. (b) A possible scheduling of (a): thread 1 acquires `x`; thread 2 acquires `y`; thread 1 tries to acquire `y` but fails as thread 2 has it acquired; thread 2 tries to acquire `x` but fails as thread 1 has it acquired. Consequently, threads 1 and 2 *block* indefinitely. That is, neither thread proceeds in its execution.

crashes, or if the host machine should be turned off for some reason [Bernstein and Goodman, 1983]. The *store* is the abstract term we give to the physical storage the transactional system interfaces with: transactions in RDBMSs interface with a store that is designed exclusively for relational data (e.g., to optimise query execution plans); by contrast, the store used by transactions in STM is shared memory. At the moment we will discard technical details and simply state that shared memory always resides in a machine's *random access memory* (RAM). The RAM of a machine is *volatile* – when a machine is turned off the contents of RAM are cleared. STM, due to the volatility of RAM, does not support durability. We will focus on STM. Under STM, if transactions issued by distinct threads access the same shared memory, and one of those accesses is a write, then one transaction will *abort* and the other will *commit*. Figure 1.12 (a) shows a DRF version of Figure 1.5. Where, `atomic { c }` executes the program commands `c`

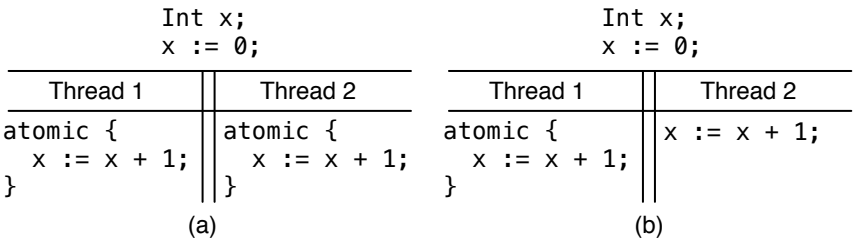


Figure 1.12: (a) Transactions are used to isolate the increments of `x` by threads 1 and 2. (b) Transactional accesses are only isolated with other transactional accesses.

under a transactional semantics. Transactions typically perform their operations on a local copy of the data they reference. This is known as *out-of-place* updates. Figure 1.13 shows a scheduling for Figure 1.12 (a). Here, each thread's respective `load` and `store` of `x` reads and writes a thread-local copy of `x`. The updates made to `x` by a transaction are only persisted to shared memory if the transaction commits. STM in many cases is also a library, so they are as prone to an error like Figure 1.9, as shown in Figure 1.12 (b). Transactional accesses are isolated only with other transactional accesses. This property is known as *weak isolation* [Harris et al., 2010].

1.1.7 Locks or Transactions?

We have presented two types of coordination so far: locks and transactions. The reader may ask why we need two types of coordination rather than just locks *or* transactions. Observing Figures 1.7 and 1.12 we note that the only difference between the program texts is the way they issue their accesses to `x`: Figure 1.7 (a) uses `sync` parameterised on a mutex; and Figure 1.12 (a) uses `atomic`. The accesses issued by a lock are isolated with respect to those issued by locks that use

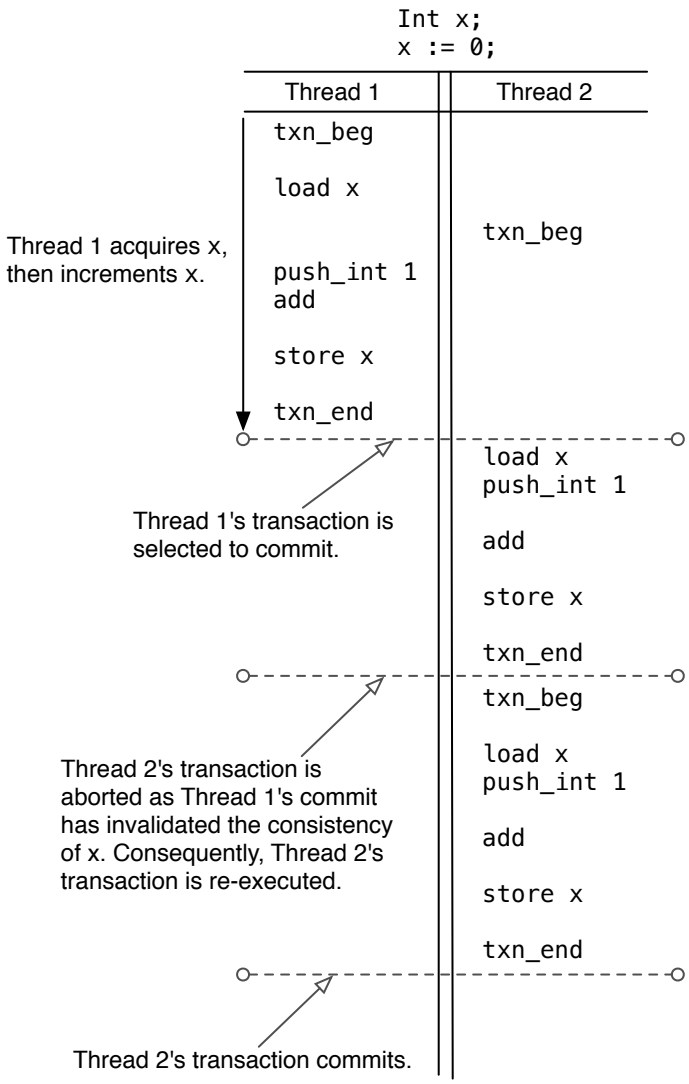


Figure 1.13: A possible scheduling of Figure 1.12 (a). `txn_beg` and `txn_end` are instructions that delimit transactional regions of program text.

the same mutex; transactional accesses are isolated with respect to those issued by other transactions. The key difference is that using a lock to coordinate accesses requires the programmer specify a mutex. In lock programming we can consider the mutex as encapsulating an *isolation invariant*. For example, we can interpret the isolation invariant of thread 1's lock in Figure 1.11 (a) as

"`acquire(v)` ∧ `acquire(w)`". The value yielded from such an expression must be *casually* true. However, the expression cannot always be casually evaluated as shown in Figure 1.11 (b). In STM isolation invariants are maintained by the STM system rather than the programmer. Consequently, STM is a lot less error-prone than locks. Furthermore, the learning curve for correctly applying locks is steep. For example, if one wishes to use locks effectively in Java, for instance, then ideally the programmer should have digested and understood the three main texts on the subject [Herlihy and Shavit, 2008; Lea, 2006; Peierls et al., 2005]. By contrast, a programmer can correctly apply STM in minutes.

We will now describe the advantages of locks and transactions, and importantly show that locks and transactions complement one another.

1.1.7.1 Pessimism and Optimism

Locks are an effective tool in the hands of an expert: they facilitate a *pessimistic*, *low-overhead* and *fine-grained* coordination semantics. By contrast, transactions are *optimistic* and simplify error-free *component composition*. Pessimistic means that for every code fragment `sync(v) { c }` the mutex `v` will *always* have been acquired before executing `c`, irrespective of whether or not `v` needed to be acquired for a given scheduling to isolate the accesses issued to `c`. For example, consider Figure 1.14 (a) where two threads read `x`. Intuitively both threads can execute their assignment concurrently without introducing a data race. However, the use of locks in this fashion always serialises their reads of `x`. The pessimism of locks in this case unnecessarily reduces the amount of concurrency that can take place. By contrast, Figure 1.14 (b) is the same as Figure 1.14 (a) but uses transactions. Here, both threads will execute concurrently because transactions

are optimistic. Conceptually one can think of the code fragment `atomic { c }` as meaning "execute `c` first and then determine if the accesses issued by `c` invalidate memory consistency." The consistency of a transaction is invalidated if it *conflicts* with another transaction. That is, two or more transactions access the same data and at least one of those transactions issues a write to that data. Optimistic coordination is more suitable than pessimistic coordination for CMPs. Furthermore, pessimistic coordination may introduce a high level of artificial contention as in Figure 1.14 (a).

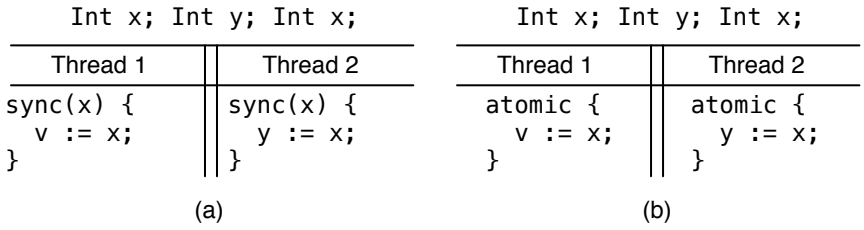


Figure 1.14: (a) Reads of `x` are always serialised due to the pessimism of locks. (b) Reads of `x` are not serialised should they be scheduled concurrently.

1.1.7.2 Overhead

The magic performed by STM does not come for free: the cost of transactionally executing commands can be great. For example, in Figure 1.13 the work performed by thread 2's transaction was thrown away due to it being aborted. The possibility of abortion is a key factor when using transactions, particularly when a transaction is accessing highly contended memory. By contrast, the cost of using a lock is generally very low and can be further reduced by using locks optimised for a particular scenario as shown in Figure 1.15. Here, three threads access `x`; thread 1 writes `x` and threads 2 and 3 read `x`. We want to isolate each

```
Int x; Int v; Int y; ReadWriteLock l;
x := 0; v := 0; y := 0;
```

Thread 1	Thread 2	Thread 3

`sync(l.WriteLock) {`
`  x := x + 1;`
`}`
	`sync(l.ReadLock) {`
	`  v := x;`
	`}`
		`sync(l.ReadLock) {`
		`  y := x;`
		`}`

Figure 1.15: Threads 1, 2 and 3 access `x`. Threads 1 and 2 only read `x` so they acquire a read lock. By contrast, thread 1 writes `x` so it acquires a write lock. Threads 1 and 2 can execute concurrently; if thread 1 has acquired the write lock then only it can execute – threads 2 and 3 will block until thread 1 releases the write lock.

access of `x` but without restricting concurrency for reads as in Figure 1.14 (a). To accomplish this we coordinate all accesses to `x` with a `ReadWriteLock l`. Thread 1 writes `x` so it acquires the write lock, `l.WriteLock`; by contrast, threads 2 and 3 acquire the read lock, `l.ReadLock`. Threads 2 and 3 can execute concurrently; however, if thread 1 has acquired the write lock then only it can execute. Another optimisation is fine-grained locking: several mutexes are used to protect possibly different regions of shared memory. Because of this greater partitioning contention is reduced, but avoiding defects such as deadlock and data races becomes harder. Figure 1.16 compares fine-grained and coarse-grained locking strategies. As a final optimisation we may combine read/write locks with fine-grained locking: this is the gold standard of applying locks; correct application of this approach is often referred to as an art rather than a science.

1.1.7.3 Composition

Designing modern systems requires composing libraries. For example, consider Figure 1.17 (a) where an item is removed from one linked list and added to another linked list. Using locks we may model a version of Figure 1.17 (a) that can

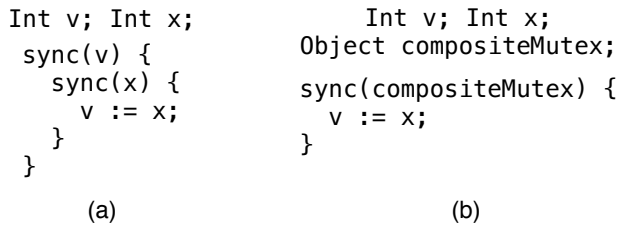


Figure 1.16: (a) Fine-grained: mutexes associated with `v` and `x` are acquired to perform the assignment. (b) Coarse-grained: a single mutex is used to protect accesses on `v` and `x`.

be performed by multiple threads as Figure 1.17 (b). The pitfall of Figure 1.17 (b) is that it is very easy to introduce deadlock and data races. Furthermore, the complexity of composing components increases as more components are composed. By contrast, transactions eliminate most of the complexity as shown in Figure 1.17 (c). Here, the STM system manages the isolation invariants to ensure that the composition is deadlock-free. Composition is the biggest advantage of STM. For example, let us consider a scenario where a programmer is asked to create a correct thread-safe version of Figure 1.17 (a). He may apply locks in several fashions and think that the solution is correct – only to observe a scheduling that invalidates his belief. Realistically the programmer would need to read and understand the locking semantics of his platform. For Java this would require him to understand Java threads [Oaks and Wong, 2004], techniques on how to correctly use locks and their auxiliary data structures [Lea, 2006; Peierls et al., 2005] and the Java memory model [Manson et al., 2005]. Often one also needs to understand the details of their host operating system's process, memory and thread scheduling internals, and in some cases the details of the underlying hardware. This is not a small undertaking. By contrast, a programmer needs only a basic familiarity with threads, concurrency and transactions to arrive at Figure

```
LinkedList l1;
LinkedList l2;

l1.add(l2.pop());
```

(a)

```
LinkedList l1;
LinkedList l2;

sync(l1) {
  sync(l2) {
    l1.add(l2.pop());
  }
}
```

(b)

```
LinkedList l1;
LinkedList l2;

atomic {
  l1.add(l2.pop());
}
```

(c)

Figure 1.17: (a) Composes the `add` and `pop` operations of the `LinkedLists` `l1` and `l2`. (b) Attempts to compose the operations in a thread-safe manner. (c) Uses transactions to safely compose the operations.

1.17 (c).

1.1.7.4 Strong and Weak Semantics

One final point of difference between locks and transactions is that locks offer a strong semantics, by contrast to transactions which are said to be weak. A lock is pessimistic which means that its protected command will always succeed: that is, in `sync(v) { c }` once `v` is acquired `c` will execute. By contrast, transactions are said to afford a weak semantics. For example, in `atomic { c }` it is possible that `c` will be executed multiple times should its transaction abort. This means that transactions are not generally safe for executing irreversible operations. Figure 1.18 shows the use of locks for writing to disk; Figure 1.19 shows a transactional version. The transactional program will guarantee shared memory consistency but not the consistency of peripheral components. It is possible, as shown in Figure 1.19, that a transaction may abort and leave some data that may be observed by subsequent reads of disk. Here, thread 2's transaction has not been atomic or consistent.

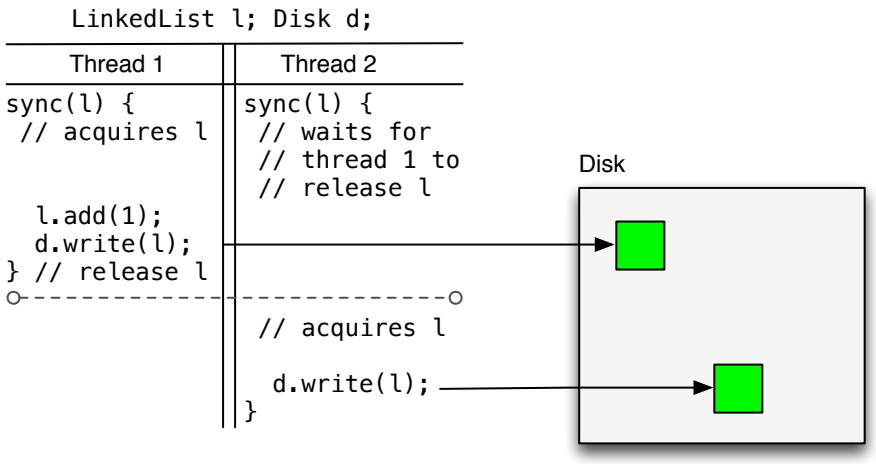


Figure 1.18: Using locks to safely execute an irreversible I/O operation.

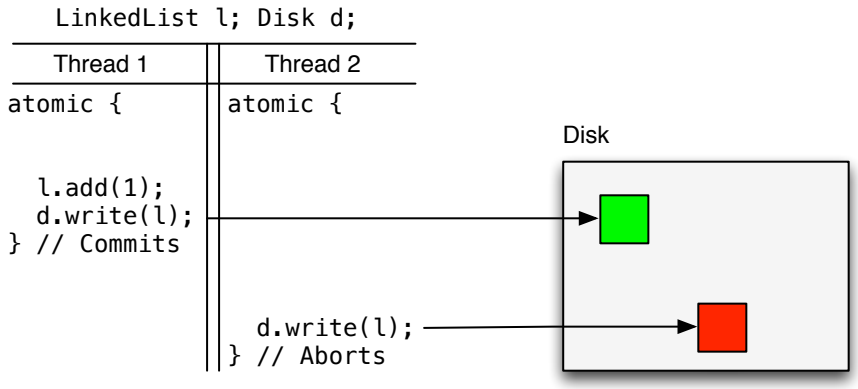


Figure 1.19: Using transactions to execute an irreversible I/O operation. Thread 2's transaction aborts but its write to disk remains. Thread 2's transaction has invalidated the atomicity and consistency guarantees.

1.2 Motivation

Correct application of locks [Dijkstra, 1968] requires a high level of programmer skill; otherwise, data races and deadlocks may be introduced. Researchers are looking into alternative methods, e.g. STM [Shavit and Touitou, 1995], to lower the barrier of entry for correctly coordinating accesses to shared memory in multithreaded programs. Adoption of STM is limited [Harris et al., 2005; Hickey, 2008] and in many cases cannot simply supplant locks (see Section 1.1.7.4). There are two key issues that are blocking the uptake of STM by mainstream imperative programming languages: (1) performance; and (2) understanding how it co-exists with existing coordination facilities such as locks. The aim of this thesis is to contribute to the literature regarding (2).

A relatively sizeable amount of literature exists on implementing transactions in systems that already expose locks, such as [Dice et al., 2006; Lev et al., 2009; Menon et al., 2008; Saha et al., 2006; Usui et al., 2009] but remarkably little exists on understanding the semantics of such systems, which is required to develop further research into the area. There are two key advantages to defining a semantic model that is based on a common implementation strategy: (1) runtime semantics; and (2) static semantics. The latter is influenced by the former: to understand what should be statically deemed correct we must understand what we wish to observe during a program's execution, e.g. [Grossman et al., 2006; Spear et al., 2008]. Most of the current literature focuses on verification of the STM system [Cohen, 2008; Guerraoui and Kapaka, 2007; Hu, 2012] or gives a semantics which focuses on a specific use case of STM [Lev and Maessen, 2005; Smaragdakis et al., 2007; Welc et al., 2008; Ziarek et al., 2008]. Moreover, the se-

mantics presented typically do not encompass several forms of coordination tools. That is, they focus on STM but omit usages of other coordination tools in the same program.

Construction of a dynamic and static semantics for programs using locks and transactions has the following concrete advantages:

- *Dynamic Semantics.* A general notion of co-existence of locks and transactions can be defined on the basis of fundamental properties such as memory locations accessed. Properties can be constructed for conflict detection and resolution between the two semantics, as well as the observational properties of reads [Adve and Gharachorloo, 1996]. With this understanding we can apply the derived knowledge to the static analysis of programs which use locks and/or transactions to coordinate accesses to shared memory. At present a clear gap in the literature exists in understanding the semantics of programs which use locks and/or transactions to coordinate accesses to shared memory.

- *Static Semantics.* Most static analysis for concurrent programs focus on programs which use locks, transactions or no coordination when issuing accesses to memory, e.g. [Boyland, 2003] and [Beckman et al., 2008]. Furthermore, most analyses that do focus on coordinated accesses – so-called "atomic blocks" abstract the concept of atomic to such an extent that they remove the practical issues faced when mixing distinct coordination semantics, i.e. locks and transactions, to attain access atomicity. Using fractional permissions [Boyland, 2003] in combination with a set of rules derived from studying the dynamic semantics of programs using locks and transactions

it will be possible to statically check their data-race-freedom.

The motivation for this thesis's work is very much exploratory: STM is currently in a state of limbo and may not see mainstream adoption; however, should it be adopted it will need to be well understood. This is particularly the case for programs that wish to use locks and transactions, as the former is ubiquitous in existing libraries which make use of multiple threads. The aim of the thesis is to shed light on this relationship so that should STM be adopted the authors of such systems have a larger wealth of literature to consult for a semantic reference.

1.3 Objectives

The objectives of this thesis are to contribute on the scarce literature that exists on programs that use distinct coordination semantics to coordinate accesses to shared memory. Specifically, this thesis focuses on the use of locks and transactions to coordinate such accesses. We have two main aims:

1. Develop a framework for reasoning about the dynamic semantics of programs that use locks and transactions to coordinate accesses to shared memory. The framework will be defined by an operational semantics. The focus of the framework is on two key elements: (i) locks and transactions; (ii) memory accesses. The semantics of locks and transactions will be derived from their respective idiomatic usages. That is, nested locks must be catered for and a conflict resolution strategy across the two semantic boundaries must be defined. A lower-level dynamic reasoning of programs that use locks and transactions will be facilitated by generalising the semantics of read/write accesses.

2. Develop a framework for reasoning about the static semantics of programs that use locks and transactions to coordinate accesses to shared memory. The framework will be defined via a set of static execution rules. Accesses to memory will be coordinated via the use of locks and transactions. A program that successfully passes the checks entailed by our framework must be data-race-free. The framework must be able to assert the data-race-freedom of a program irrespective of whether the same memory is accessed transactionally, via a lock or under no coordination semantics. A program which fails our static framework is not data-race-free.

Both frameworks must be able to model reasonable usages of locks and transactions in a multithreaded program. However, each framework will focus on the relevant use cases. At the present time it is not tractable to reason about arbitrary multithreaded programs that use locks and transactions to access shared memory.

1.4 Challenges

The following challenges exist to successfully meet the objectives of this thesis:

- STM is not like locks: a consensus does not exist on what the semantics of STM should be. A semantics will have to be defined based upon the commonality of the existing implementations of STM.

- Locks and transactions differ in how isolation invariants are defined, and what those invariants mean. In STM isolation invariants are accumulated optimistically whereas a lock's invariant is specified pessimistically. A key

issue will be defining how the invariants of locks and transactions can be preserved without violating the semantics of either a lock or transaction. The conflict strategy that is chosen should not restrict concurrency unless programmer specified isolation invariants dictate otherwise.

- The dynamic semantics should permit reasoning about a program that uses locks and transactions to the level of individual read and write accesses. This will facilitate the generalisation of observation properties so that we can define properties based on their semantics and map them to existing memory consistency models.

- The static semantics should successfully identify program accesses that may result in a data race and correctly classify programs that issue such accesses as not being data-race-free. Classification of a program's data-race-freedom should be based upon reads and writes to memory locations that are inline with the dynamic semantics. That is, the object model should be that of struct semantics in C. The static analysis should not be overly conservative. For example, distinct threads that access distinct fields of the same object should not be flagged as inducing a possible data race.

Some of these challenges will restrict the amount of work that can be done, particularly for our static semantics which are not envisioned to be able to address the data-race-freedom of programs using a large array of program features. The current literature suggests that such a task at present is not feasible for programs that use a single coordination semantics, let alone one that uses two coordination semantics that differ to the extent of locks and transactions.

1.5 Contributions

This thesis presents three main contributions Barnett and Qin [2012a,b, 2013] which fall under one of two domains: *dynamic reasoning*, covered in Part I; and *static reasoning*, which is covered in Part II of this thesis. In summary, the contributions presented in this thesis are:

- *Moverness* Barnett and Qin [2012a], a correctness criterion for modelling locks and transactions in memory consistency models. We find locks to be *left movers*, transactions *right movers*, transactions and locks with respect to themselves *both movers* and non-conflicting locks and/or transactions *free movers*. Moverness trivialises reasoning about the otherwise complex semantics of locks and transactions, particularly in programs which use both to coordinate accesses to shared memory. We validate moverness by giving a case study showing its mapping to the happens-before memory consistancy model used by the Java memory consistancy model. Our definitions of moverness are successful if it faithfully encodes the semantics of the happens-before memory consistancy model. To our knowledge moverness is the first correctness criterion for encoding locks and transactions in memory consistancy models.

- *Guaranteed Transactions* Barnett and Qin [2012b], a semantic construct encapsulating the privatisation and publication idioms. Guaranteed transactions are pessimistic and non-abortable but maintain a transactional interface. We validate guaranteed transactions by giving a case study based upon Spear et al. [2007] that aborted in-place updates are never observed, and that out-of-place committed updates are always observed. Our suc-

cess criteria is by showing the ommission of the former anomolies during the use of the guaranteed transactions. We also formulate the meaning of guaranteed transactions under moverness. Guaranteed transactions are an enhancement over existing pessimistic transactions, while not precluding non-conflicting guaranteed transactions to execute concurrently.

- *Data Race Freedom* Barnett and Qin [2013], a static analysis framework for determining whether a program entailing locks and transactions is data-race-free. Our static framework entails two general stages: first, the program is statically executed in order to characterise the reads and writes it issues; then, an isolation algorithm determines the isolation of accesses issued by the program to each region of memory it allocates. A program that satisfies our isolation algorithm is data-race-free. We validate our framework by applying to a series of case studies entailing a number of non-trivial programs, including ones which access dynamically allocated memory. The success criteria for our static analysis is by such the presence or respectively the omission of programs which exhibit and respectively do not exhibit data races. To the best of our knowledge our static analysis is the first to guarantee the data-race-freedom of programs that entail locks and transactions.

Chapter 2

Literature Review

This chapter presents a survey of the literature the work presented in subsequent chapters is related to. The related work can be partitioned into the following three groups:

1. programming languages;
2. locks and software transactional memory (STM); and
3. memory consistancy models.

These three groups comprise an authoritative survey of concurrency in current and state of the art environments: programming languages are often coloured by the synchronisation and concurrency features built into the language (e.g., Erlang Armstrong et al. [1996] with its threads and actors, typed channels in Google Go Google-Go [2013], and `synchronized` in Java Arnold et al. [2005], etc.); locks [Dijkstra, 1968] and transactions [Shavit and Touitou, 1995] are two semantics that synchronisation primitives may reduce to (the focus of this thesis); finally, all

synchronisation facilities must have an established meaning in the memory consistancy model Adve and Gharachorloo [1996] of the respective language/runtime (e.g. Java [Manson et al., 2005] and C++11 Boehm and Adve [2008]. That is, there must be a systematic way to reason about and relate accesses issued to memory by distinct threads.

The literature presented here gives the general positioning of the work which follows later in this thesis. Future chapters position their respective work explicitly with respect to the work we now cover. The first section on programming languages gives a general overview of innovation in programming language technologies, libraries and ancillary services with respect to concurrency and coordination. Subsequent sections on locks and transactions and memory consistancy models which are of most import to the work presented in this thesis. Special attention is given to the semantics of locks and transactions and the current literature which reasons about such programs.

2.1 Programming Languages

In this section we trace the roots of cutting edge concurrency idioms encoded in today's programming languages. Several languages give innovative treatments of concurrency, a non-exhaustive overview includes: Cilk [Blumofe et al., 1995] – a famous MIT project that popularised *spawning* threads and *cactus stacks*; Erlang [Armstrong et al., 1996] – born out of Ericsson for programming highly reliable hardware such as switches; Haskell [, editor] – which encodes parallel and synchronisation idioms with the assistance of its expressive type system; and Clojure [Hickey, 2008] – that introduced persistent data structures and STM to

the Java enterprise. The big industry innovators have also inflicted their idea of how they believe concurrency should be done: Google designed Go [Google-Go, 2013], a language that uses message passing [Hoare, 1978]; Microsoft has concurrency platforms for C++ and .NET, and an impressive extension of C++ that allows programmers to easily program graphics processing units [Microsoft, 2013a]; Intel has contributed an efficient version of STM for C++ [Intel, 2012] and a C++ variant of the Cilk MIT project, Cilk Plus [Intel, 2013b]. NVidia has opened up their GPUs via CUDA C [Farber, 2011]; and most recently Mozilla has began developing Rust [Mozilla-Rust, 2013], a language that uses affine/linear types to guarantee data is safely shared among threads. The rest of this section describes some of these languages and their key innovations.

2.1.1 Threads and Tasks

2.1.1.1 Threads

A fundamental aspect of concurrency is understanding that multiple things can happen at the same time. In modern programming environments this concept is facilitated by threads and tasks. In general the semantics of threads are uniform across programming languages, with the exception of their *abstract programming interfaces* (APIs). By contrast, tasks were popularised by Cilk [Blumofe et al., 1995] and provide an efficient means to schedule large amounts of concurrent units of work. At the most basic level threads and tasks differ in their resource profile: creating a thread requires the operating system allocate (a relatively large) amount of memory for the thread's stack, typically around $1-2$ megabytes; by contrast, tasks consume little more resources than an object. Threads are

scheduled by the operating system's thread scheduler; by contrast, tasks are scheduled to threads by a *task scheduler* [Blumofe and Leiserson, 1994].

Most programming languages expose threads through an API that closely resembles that of the underlying system interface, e.g. Win32 [Russinovich et al., 2012] and POSIX [Butenhof, 1997]. Typically a thread API offers the following abilities: thread creation; assignment of some program text the thread is to execute; and ways to start, wait for and cancel the thread. We will not discuss safe thread cancellation as it differs per program text the thread is executing. The excellent texts [Peierls et al., 2005] and [Lea, 2006; Oaks and Wong, 2004] provide a wealth of practical advice on thread cancellation and multithreading in general, all be it specific to Java [Arnold et al., 2005]. For .NET programmers the standard texts are [Duffy, 2008; Richter, 2012] and for C++ there is [Williams, 2012]. Specifying the program text a thread should run varies according to the programming environment: pthreads [Butenhof, 1997] take a pointer to a function; by contrast, languages such as Java [Arnold et al., 2005] and C#[Hejlsberg et al., 2010] permit the programmer richer interfaces such as `java.lang.Runnable` in Java or lambda expressions/delegates for .NETs `System.Threading.Thread` type. Most threading APIs support *thread local storage (TLS)*: the ability for a thread to allocate and access memory that only it can access. For example, in D [Alexandrescu, 2010] all data by default is in TLS, and in .NET one can use the `ThreadStatic` attribute to denote that the data it decorates should be stored in TLS. The programming models that we use in this thesis are all based upon the use of threads at an abstract level and are implementation agnostic. Furthermore, we assume a perfect environment where if a program defines N threads then there exists N PEs to execute such threads.

2.1.1.2 Tasks

Tasks are an abstraction of threads specifically to support the effecient modelling of large amounts of concurrent work. Figure 2.1 gives a high-level architectural overview of a task system. Cilk [Blumofe et al., 1995] inspired most of the task-based libraries that exist today, including Intel Threading Building Blocks [Reinders, 2010] and Microsoft's C++ Concurrency Runtime [Microsoft, 2013b] and Task Parallel Library [Microsoft, 2013c] for .NET. There are a few key architectural properties of task-based runtimes, which we will now describe. Figure 2.1 shows a *task scheduler*. The task scheduler is a *user mode* [Bovet and Cesati, 2005; Kerrisk, 2010; Russinovich et al., 2012] component, that is it lives outside of the operating system's kernel, *kernel-mode*. The task scheduler is designed to be able to schedule huge numbers of tasks by multiplexing them onto a finite number of threads which the task runtime creates. For example, in Figure 2.1 the task runtime creates three threads, and subsequently maps the tasks created by the process to those three threads. Most task schedulers in use today employ some form of *work stealing* [Blumofe and Leiserson, 1994]. That is, the task scheduler can steal tasks it assigned to one thread and map them to another thread. Erlang supported Cilk-like tasks and scheduling for symmetric multiprocessor systems since Erlang R11B released in 2006. Apple also has a task technology known as *grand central dispatch (GCD)* which can be utilised by programs targeting OSX or iOS. Under GCD the work a task is to perform is encapsulated within a *block* in Objective-C [Kochan, 2012] which is similar to a closure, or block in Ruby [Flanagan and Matsumoto, 2008]. OCaml [Leroy et al., 2012] has something similar to threads but under the guise of a user contributed

light weight threads library, *lwt* [Dimino, 2012].

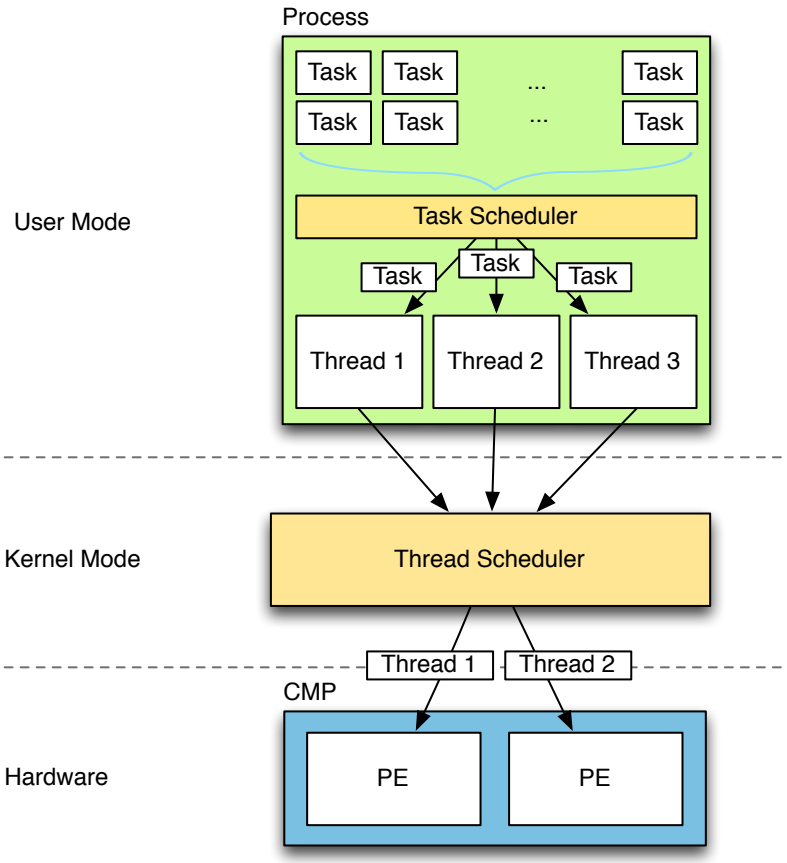


Figure 2.1: High-level architecture of a process that uses tasks.

2.1.2 Immutability

One of the key tenants of being able to reason about concurrent programs is *immutability*. An immutable data structure never changes and is thus free from being subject to a data race [Unger, 1995]. Functional languages such as Haskell [, editor] and OCaml [Leroy et al., 2012] are immutable by default, although in OCaml mutating data can be done when necessary. The level of immutability

supported in languages such as Java, C++, C and C# is relatively weak. For example, in C++ [Stroustrup, 2000] application of `const` can result in immutable semantics but requires a great deal of design attention; in C# `const` is much weaker than C++'s `const`, consequently `readonly` is used but again the use of `readonly`, like C++'s `const`, requires great deal of attention to design immutable structures. One interesting approach to immutability by an imperative language is that taken by D [Alexandrescu, 2010] which has an `immutable` modifier. In D any data that has the `immutable` modifier is immutable, where immutability spans the transitive closure of the reachable object graph for that data. Scala [Odersky et al., 2011] and Rust [Mozilla-Rust, 2013] support immutability of varying strengths by default. For example, in Scala the standard modifier to use for data is `val` which denotes an immutable value, however the data reachable through a `val` may be mutable. In this thesis all data structures are mutable. We are interested in situations when data races can be introduced so we explicitly force the programmer down the road of mutation.

2.1.3 Memory Management

There are two types of memory management: *deterministic* and *non-deterministic*. Determinism in the context of memory management determines when memory will be recycled for use by other requests to the memory manager, e.g. through calls to `malloc` in C [Ritchie and Kernighan, 1988] or `new` in C++ [Stroustrup, 2000] and Java [Arnold et al., 2005]. C and C++ are deterministic: deallocation of heap data is immediate and performed at a point of the programmer's choosing. For example, in C++ one would allocate data on the heap by `new` and then

subsequently delete the memory allocated by `new` by either `delete` or `delete[]`. In C++ one can also use `shared_ptr`, `unique_ptr` and `weak_ptr` types to assist in the lifetime of heap data, but deallocation remains deterministic [Josuttis, 2012]. Non-deterministic memory management is typically employed by higher-level languages such as Java [Arnold et al., 2005], C# [Hejlsberg et al., 2010], Haskell [, editor] and OCaml [Leroy et al., 2012], to name just a few. These environments are non-deterministic as it is the *garbage collector (GC)* [Jones and Lins, 1996; Jones et al., 2011] that determines when heap memory is to be recycled, not the programmer. The performance of GCs varies but in general they are slower than the deterministic deallocation of C and C++. From our perspective the main advantage of a GC is that it makes memory management in concurrent programs a great deal simpler and safer. For example, a GC is almost always required to implement persistent data structures [Okasaki, 1996] correctly. Most recent environments that admit multi-threaded programs employ a GC, e.g. the Java Virtual Machine (JVM) [Lindholm et al., 2013] and the Common Language Runtime (CLR) [Richter, 2012]. The use of a GC makes concurrent programming much simpler as the lifetime of memory is deferred to the GC rather than the programmer. Memory management is not a key component of the work presented in this thesis but we assume that allocated memory is implicitly reclaimed.

2.1.4 Message Passing

Message passing [Hoare, 1978] is a type of *coordination*. Examples of language support for message passing includes Erlang [Armstrong et al., 1996], Google's Go [Google-Go, 2013] and Mozilla's Rust [Mozilla-Rust, 2013] programming lan-

guages. Other languages also support message passing but via libraries, e.g. Scala [Odersky et al., 2011] whose message passing library is based upon Akka [Type-Safe, 2013] and Haskell's recent Erlang-like library which is discussed in [Epstein et al., 2011]. We do not cover message passing in this thesis.

2.1.5 GPGPU

General purpose graphics processing units (GPGPU) are becoming ubiquitous. The two market leading GPU manufacturers – AMD and NVidia – both support GPGPU. That is, it is possible to run general purpose computations on AMD and NVidia hardware, which is otherwise the domain of graphics-specific computations. AMD and NVidia provide proprietary software development toolkits for programming their respective GPU hardware, such as NVidia's CUDA [Farber, 2011; Sanders and Kandrot, 2010] which is typically driven by a variant of C known as Cuda-C. In addition to the proprietary toolchains there is also an effort to provide libraries and tools for standards conforming languages such as C++. For instance, AMD has recently released the Bolt library; by contrast, NVidia has its Thrust library. Both Bolt and Thrust [Farber, 2011] have similar interfaces to the C++ standard template library (STL) [Stepanov and Lee, 1995]. However, unlike STL, both Bolt and Thrust perform their computations on the discrete GPU. Microsoft has also tried to aid C++ programmers by extending the C++ language with a set of language features that specifically target the discrete GPU should the host have one, known as C++ Accelerated Massive Parallelism, or simply C++ AMP [Microsoft, 2013a]. In this thesis we focus on the traditional computation architecture comprising a CMP and a set of of memory modules

(shared memory) the CMP directly accesses.

2.2 Locks and Transactional Memory

Locks and transactional memory are used to facilitate *mutual exclusion*. The operations of two threads are mutually exclusive if only one thread can issue accesses. Locks and transactions are the primary focus of this thesis. The observation in the following literature is that little work exists on the theoretical underpinnings of programming models that permit both locks and transactions to coordinate accesses to shared memory. Subsequent chapters will focus on addressing this ommission in the current literature. For convenience, before exploring locks and transactions, Table 2.2 shows the coordination tools used by a select number of programming languages.

Language	**Functional/Imperative**	**Coordination Semantics**
C#	Imperative	Locks
Java	Imperative	Locks
C/C++11	Imperative	Locks
Erlang	Functional	Message passing
Google Go	Imperative	Message passing and locks
Haskell	Functional	Locks and STM
Clojure	Functional	Locks and STM
D	Imperative	Message passing and locks

Table 2.1: Coordination control used in select programming languages.

2.2.1 Locks

Locks [Dijkstra, 1968; Hoare, 1974] are a facility to limit the number of threads that execute a particular region of code concurrently. A *semaphore* permits N

threads to execute a region of code. A semaphore where $N = 1$ is a *binary semaphore*, most often referred to as a *mutex*.

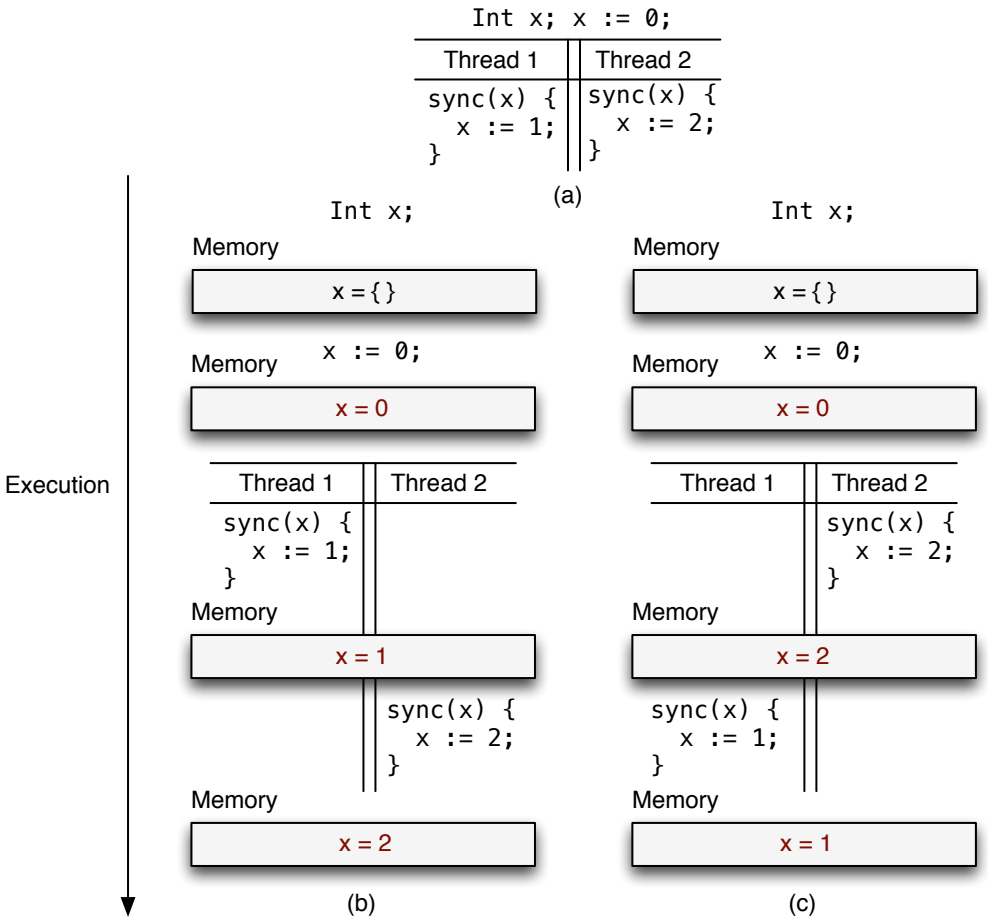


Figure 2.2: (a) `sync(x) { ... }` denotes an explicit lock protected on `x`. Two threads update the value of `x`; each update is protected on the mutex associated with `x`. (b) and (c) show the possible thread schedules.

In Java the semantics of an *implicitly synchronised* `synchronized` block is that of a mutex. That is, if one has as part of a class definition in Java a method with the signature `synchronized void mutate() { ... }`, then only one thread can execute `mutate` at a time for a given object. For example, in `o.mutate(); || o.mutate();` a total ordering is enforced over the invocations of `mutate` should

they be scheduled concurrently. Semaphores and mutexes are supported in most programming languages and libraries. Like Java, C# also gives language support with `lock` for using locks but not at the method interface level. Instead, in C# one always uses *explicit synchronisation*. Explicit synchronisation is where the programmer explicitly parameterises the object we wish to delegate mutual exclusion to. For example, `synchronized(this)` { ... } is a form of explicit synchronisation in Java, despite it yielding the same semantics as our `mutate` method if it encapsulated the whole of the method's program text. A similar approach can also be taken in C# but using `lock`, although this is idiomatically incorrect. In C# one often provides a property that yields a thread-safe object that clients can synchronise on. This can be observed by the types in `System.Collections`. In both Java and C# every object has an associated lock. The lock resides in the object header and is lazily initialised upon its first acquisition [Stutz et al., 2003]. When one uses an implicit lock or explicit lock parameterised on `this` in Java, it is the object lock we are acquiring. The formal name for this type of lock in Java is known as a *monitor* [Arnold et al., 2005]. For the purposes of this thesis we simply treat a monitor as a mutex, despite a monitor facilitating thread rendezvousing via its `notify`, `notifyAll` and `wait` methods defined in `java.lang.Object`.

Figure 2.2 gives an example of using explicit synchronisation. Here, there is a total ordering over the updates of `x` should they be scheduled concurrently as both updates of `x` are protected on the same mutex. A thread must have *acquired* the mutex `x` before entering its critical region. When a thread exits its critical region it *releases* `x`. Only one thread can acquire `x` at a time. If we were to use a semaphore with N participants then N threads could acquire the semaphore.

Figure 2.3 shows a possible scheduling of acquire/release events that result in Figure 2.2 (b).

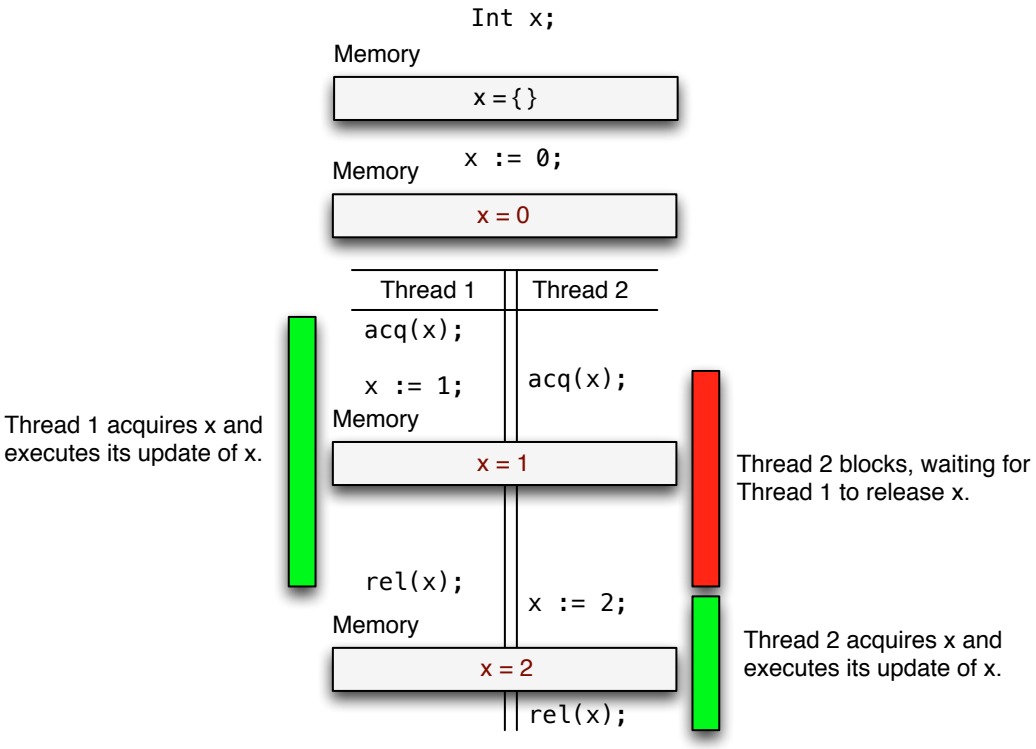


Figure 2.3: A possible scheduling that leads to the ordering in Figure 2.2 (b). We use the pseudo instructions `acq` and `rel` to denote acquire and respectively release operations of the mutex associated with `x`.

Mutual exclusion when locks are protected on a mutex is only guaranteed should both locks use the same mutex. Figure 2.4 shows Figure 2.2 (a) but differs in that both locks use a different mutex to protect their write of `x`. For Figure 2.4 (a) three possible schedules exist: that of (b) and (c) in Figure 2.2 and (b) in Figure 2.4 where the updates may take place concurrently result in a data race [Unger, 1995] on `x`. In Figure 2.4 (b) a data race can materialise in a similar manner to that demonstrated in Figure 1.6.

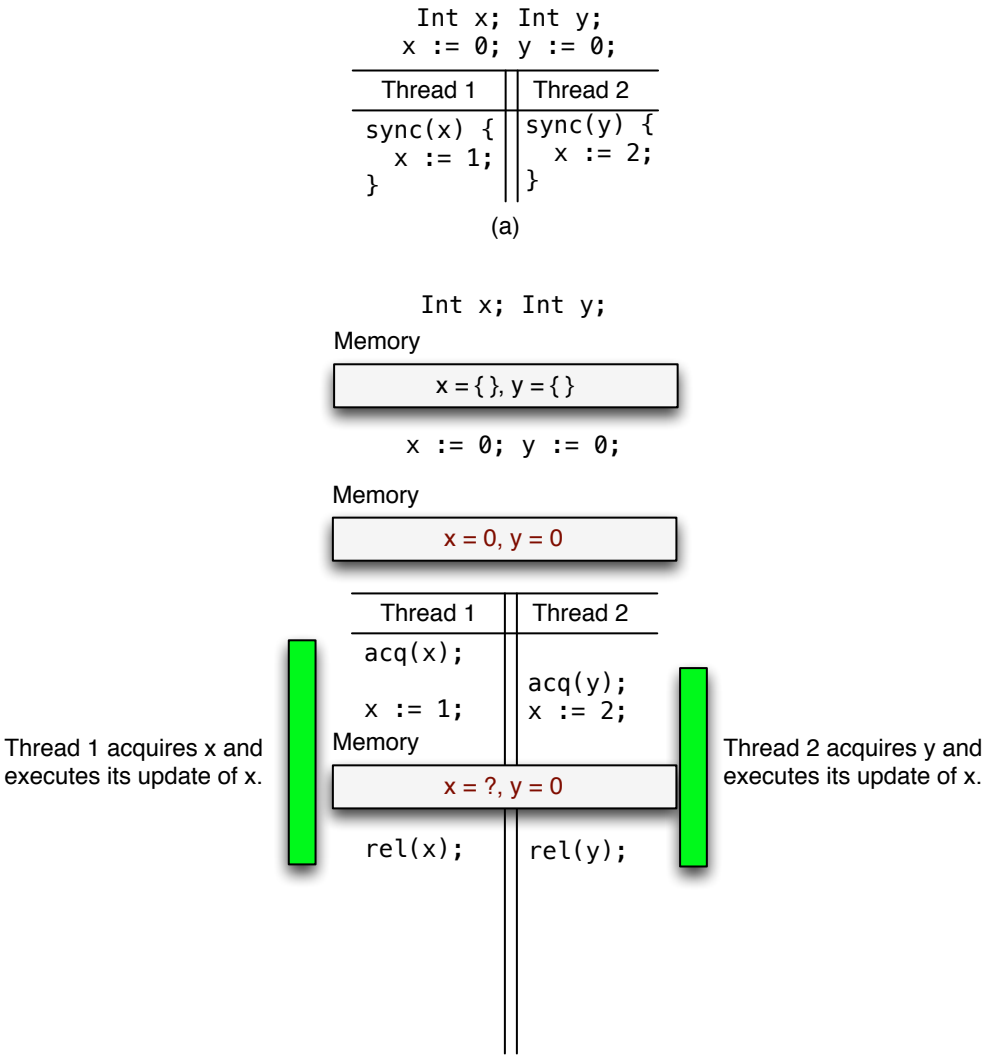


Figure 2.4: (a) The writes of `x` are protected on different mutexes. (b) A possible scheduling of (a). Each thread's write of `x` can occur concurrently, leading to a data race on `x`.

Mutexes, semaphores and so on, are required to be acquired in a consistent order. In most languages this order is not encoded in the programming language's type system or runtime semantics. The programmer must remember the order of acquisitions when he or she wishes to access data that is shared between threads. The standard convention is to document such orders within the program text in

the hope that maintainers of the software will adhere to such advice. Lock acquisition order is important because it may lead to a situation known as *deadlock* [Zöbel, 1983]. A contrived but simple example of deadlock is given in Figure 2.5. The immediate observation in Figure 2.5 is that each thread in (a) acquires the mutexes `x` and `y` in the opposite order with respect to the other thread. Figure 2.5 (b) shows one potential scheduling of Figure 2.5 (a). Here, thread 1 acquires `x` then thread 2 acquires `y`. After the each thread's initial mutex acquisition they wish to acquire the mutex that is held by the other thread. Since this is not possible, as only one thread can acquire a mutex, both threads make no further progress in their respective program text's. Deadlock, like data races, are a common occurrence in concurrent programs, particularly larger software where acquisitions and releases are hidden behind layers of indirection. The subjective opinion of the author is that deadlock is an easier problem to reason about than data races. Deadlock can be apparent in many cases. Attaching a debugger to a program you believe to be subject to deadlock can easily confirm your suspicion. By contrast, data races seldom give any clue to their presence.

Reasoning about concurrent programs that use locks or no coordination has been the focal point of most of the current literature. A few of the most prominent practically used dynamic analyses for concurrent programs include Helgrind [Valgrind-Project, 2013] (a tool in the Valgrind [Nethercote and Seward, 2007] suite) and Google's ThreadSanitizer [Serebryany and Iskhodzhanov, 2009]. Note that ThreadSanitizer is also a tool which is to be used with Valgrind. Both tools use happens-before [Lamport, 1978] (discussed in Section 2.3) to establish the relative ordering of accesses. Helgrind is largely tied to programs that exlusively use pthreads [Butenhof, 1997]. By contrast, [Serebryany and Iskhodzhanov, 2009]

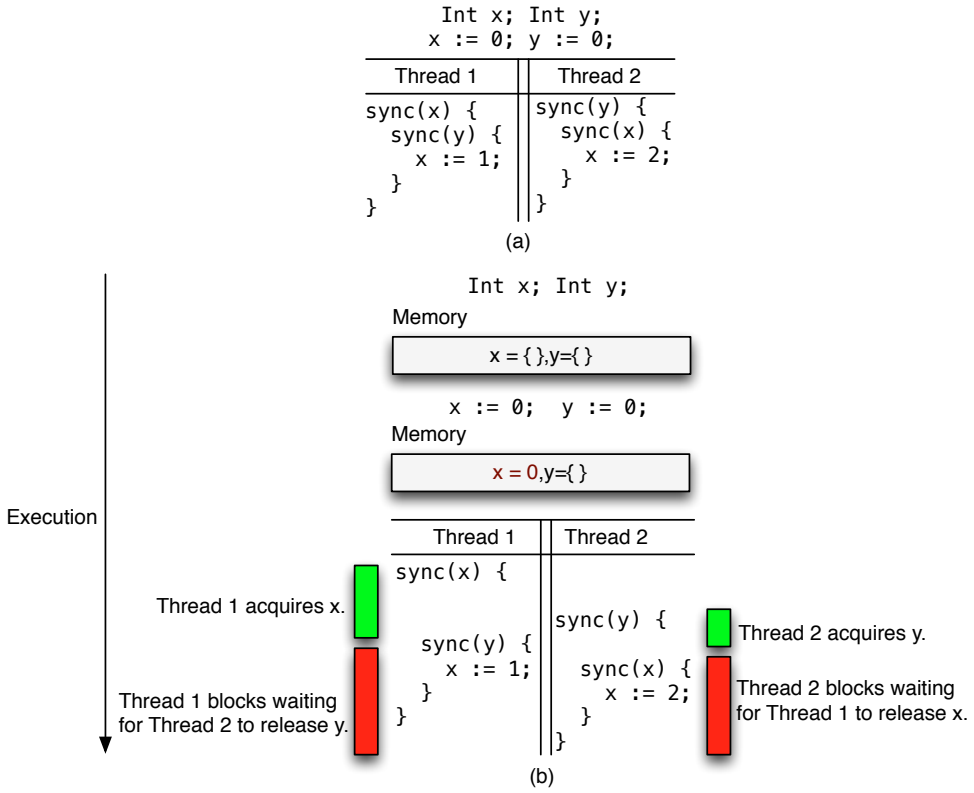


Figure 2.5: (a) Each thread acquires the mutexes associated with `x` and `y` in the opposite order to the other thread. (b) A possible schedule that leads to deadlock. Here, thread 1 acquires `x` then thread 2 acquires `y`. Neither thread can make any progress as each thread is waiting on the other thread to release their mutex. For this scheduling the value of `x` will remain 0.

provides a set of annotations that permit the programmer to direct the dynamic analysis of concurrent programs that do not use pthread coordination primitives. ThreadSanitizer is used to check the data-race-freedom of the open source Chromium Browser [Chromium-Project, 2013]. The Google Go [Google-Go, 2013] programming language, as of version 1.1, comes with a data race detector tool that is based upon ThreadSanitizer.

Fractional permissions [Boyland, 2003] can be used to facilitate a simple and intuitive partitioning of the reads and writes a program issues. This is particularly

helpful when reasoning about concurrent programs. For example, concurrent reads to the same memory are inherently data-race-free, but concurrent accesses where at least one of those accesses is a write, are not data-race-free. Under fractional permissions rationales are used to classify the type of access issued the program text. For example, given the command `x := v` we have a read of `x` and a write of `y`. Using fractional permissions we may represent these accesses as: 1`x` and ϵ `y`, where 1 (a whole) represents a write and $0 < \epsilon < 1$ represents a read. Using basic addition we can add these so-called fractions on memory locations to determine whether or not coordination is required to prevent data races. A lot of the recent literature on verifying concurrent programs use fractions in some form, e.g. [Bornat et al., 2005] and [Heule et al., 2011], the latter of which is used in the verification tool Chalice [Leino et al., 2009]. We also use fractional permissions as the basis for the analysis we present in Part II of this thesis.

2.2.2 Software Transactional Memory

"Atomic" blocks were described first by [Lomet, 1977]. Later, *Transactional memory* (TM) was proposed as a set of hardware extensions by [Herlihy and Moss, 1993]. *Hardware transactional memory* (HTM) has made some progress since its introduction, most notably with Sun Microsystems's ROCK server CMP [Chaudhry et al., 2009], but was promptly cancelled before it could gain any traction. As the name implies HTM is a variant of TM that requires hardware support. The advantage of HTM is that it is a great deal faster than software emulated TM; its disadvantage is that it requires CMPs with HTM support to saturate the market before TM can be a viable programming model. We will focus

on software emulated TM: *sofware transactional memory* (STM) [Shavit and Touitou, 1995], which came after the innovations of [Lomet, 1977] and [Herlihy and Moss, 1993].

STM has gained a considerable amount of traction over the last decade, with both language [Harris et al., 2005; Hickey, 2008] and library [Dice et al., 2006; Saha et al., 2006] support. The thesis of STM (and TM) is simple: instead of locks we wish to use transactions [Bernstein and Goodman, 1983] to isolate accesses to shared memory. Transactions in a *relational databased management system* (RDBMS) guarantee the *ACID* properties:

Atomicity The *effect* of the transaction appears to take effect as a single *indivisible* step, or not at all.

Consistancy The data in the *store* is contributed only by transactions which *commit*. A transaction that *aborts* never contributes its effect to the store.

Isolation The effect of transactions are isolated with respect to other transactions.

Durable The effect of committed transactions and by extension the consistent store is *persistant*. That is, the store may be rehydrated in the case of a hardware failure. Durability in modern RDMBSs is often facilitated by *replication* [Microsoft, 2012].

We will now refine the previous terminology for STM and HTM which only support the *ACI* properties of the *ACID* acronym. We will explain the terminology used in the descriptions later.

***A*tomicity** The *effect* of the transaction appears to take effect as a single indivisible step, or not at all. The effect of a transaction may be *in-place* or *out-of-place* [Harris et al., 2010].

***C*onsistancy** The data in memory is contributed only by transactions which *commit*. A transaction that *aborts* never contributes its effect to memory.

***I*solation** The accesses issued by transactions are at the very least isolated with respect to other issued by other transactions, known as *weak isolation*; TMs that isolate transactional accesses with non-transactional accesses are *strongly isolated* [Harris et al., 2010]. Most STMs are weakly isolated; HTMs typically afford a strongly isolated semantics, although research has been conducted on bringing strong isolation to STM [Abadi et al., 2009].

The semantics of STMs pivot on several components, generally they are:

Granularity of Conflict Detection . Variants include address-based, object [Harris et al., 2010] or more abstract, e.g. linearisability [Herlihy and Koskinen, 2008; Herlihy and Wing, 1990; Koskinen et al., 2010].

Update Mode In-place [Moore et al., 2006] or out-of-place [Harris et al., 2010]. In-place transactions mutate the memory they access in-place; out-of-place transactions issue accesses to a copy of their data [Harris et al., 2010].

Contention Management The contention manager decides which transactions should abort and commit should the same memory be contended by several transactions. Typically contention management employs a heuristic that is domain specific, just like an operating system's thread scheduler or a task scheduler [Herlihy et al., 2003; Spear et al., 2009].

Isolation The level of isolation afforded by STM is typically weak isolation: transactional accesses are isolated only with other transactional accesses. Strongly isolated STMs isolate transactional accesses with transactional and non-transactional accesses [Harris et al., 2010].

Nesting Transactions can be open, closed or flattened.

The remainder of this section dissects the properties of STM which are relevant to the work presented in this thesis.

2.2.2.1 Basics

We now give an abstract overview of transactions in STM. In particular we focus on STM in relation to the general abstractions encoded by the ACI properties. The ACI properties will be expanded upon in subsequent sections.

Figure 2.6 gives a diagrammatic representation of a transaction's structure with regards to memory accesses. Each command of a transaction issues a sequence of reads and writes to memory. The set of memory locations a transaction reads is known as its *read set*; those that it writes form the transaction's *write set*. A transaction's *dataset* is the union of its read and write set.

Two transactions *conflict* if the write set of one transaction intersects the dataset of another transaction. Figure 2.7 show two scenarios: (a) when transactions do not conflict; and (b) when they do conflict. Only one transaction may *commit* should there be a conflict. The transactions that do not commit must *abort*. The transaction that commits contributes its effect to memory. The aborted transactions do not contribute their effect to memory. Each aborted transaction is re-executed. The thread that executed the committed transac-

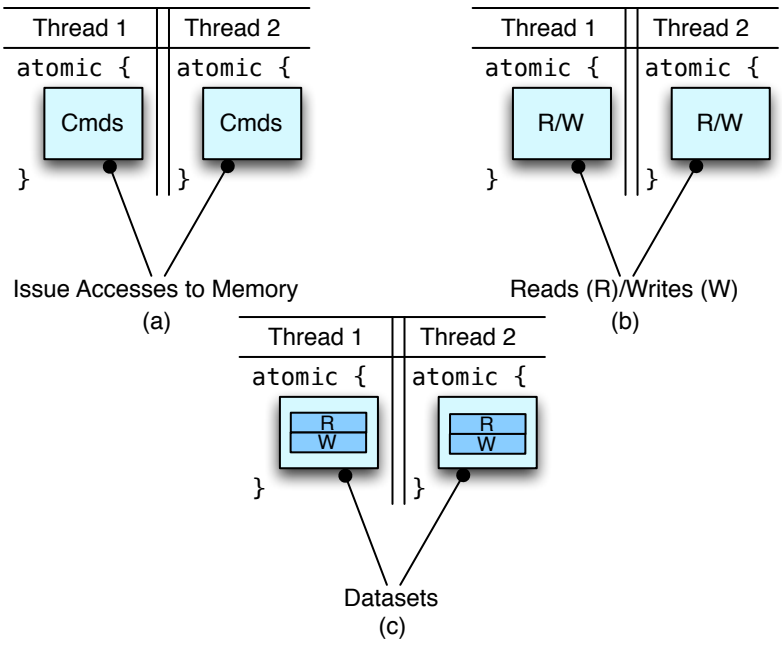


Figure 2.6: Abstract view of transactional accesses to memory. (a) A transaction entails a number of commands to execute. (b) Each command to be executed by a transaction issues a sequence of reads and writes to memory. (c) The set of memory locations a transaction accesses is known as its *dataset*.

tion proceeds by executing its subsequent program text. Figure 2.8 shows the commit/abort semantics of transactions.

2.2.2.2 Isolation

In Figures 2.7 and 2.8 we described the notion of conflict. We will now discuss the types of accesses that transactional accesses may conflict with, known as *isolation*. TM employs one of two types of isolation: *weak* or *strong* [Harris et al., 2010]. Weak isolation is prevalent in STM [Dice et al., 2006; Hickey, 2008; Menon et al., 2008], however some languages such as Haskell [, editor] exploit the type system to give a semantics similar to strong isolation [Harris et al., 2005].

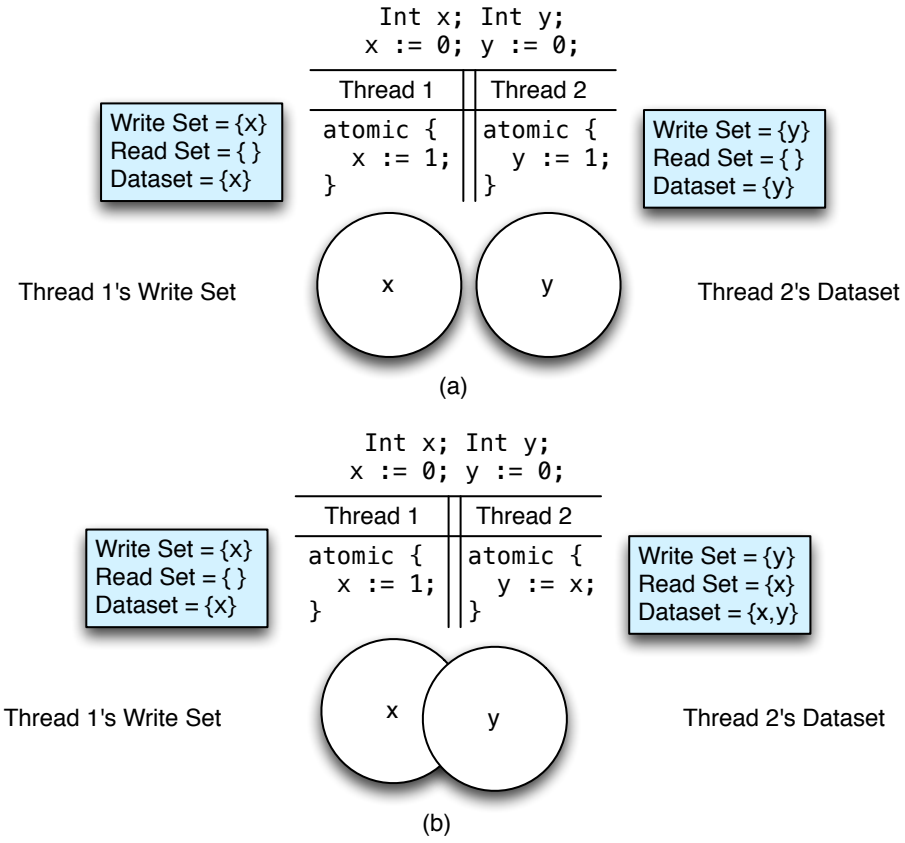


Figure 2.7: (a) The write set of thread 1's transaction does not intersect with the dataset of thread 2's transaction. (b) The write set of thread 1's transaction intersects: only one of the two transactions may commit.

The accesses issued by a transaction in a weakly isolated STM are isolated with respect to other transactional accesses. Figure 2.9 show a weakly isolated semantics. In (a) the final value of `y` will be either 0 or 1, due to its read of `x` being ordered before or after thread 1's write of `x` (see Figure 2.8 as to why). By contrast, the final value of `y` in (b) will be 0, 1 or a junk value which we simply label ?. In a weakly isolated STM transactional accesses are only isolated with other transactional accesses. Therefore, there does not exist a total ordering over the transactional write of `x` by thread 1 and the uncoordinated read of `x`

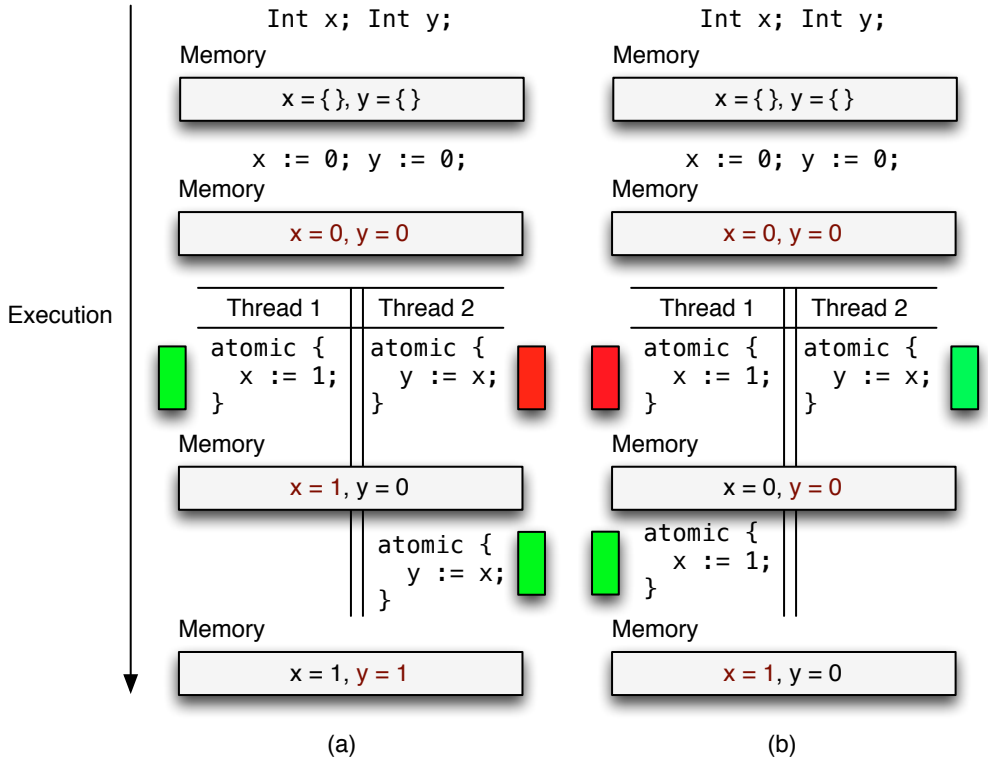


Figure 2.8: (a) Thread 1's transactional write of `x` is selected to commit. Thread 2's transactional read of `x` is aborted and subsequently re-executed, upon which it observes 1 for the value of `x`. (b) Is the reverse of (a). Thread 2's transactional read of `x` observes 0 as its value. Thread 1's transactional write of `x` is aborted and subsequently re-executed.

in thread 2. If the thread scheduler executes thread 2's read before thread 1's transactional write of `x` then the final value of `y` will be 0; if scheduled after then its final value will be 1. However, if both thread's accesses of `x` are scheduled concurrently, thread 2's read of `x` could observe an intermediate value, a so-called *junk value*. A junk value is a side-effect of a data race. See Figure 1.6 for the intuition behind how such a junk value may occur.

Strong isolation goes one further than weak isolation and not only guarantees that transactional accesses are isolated with respect to other transactional

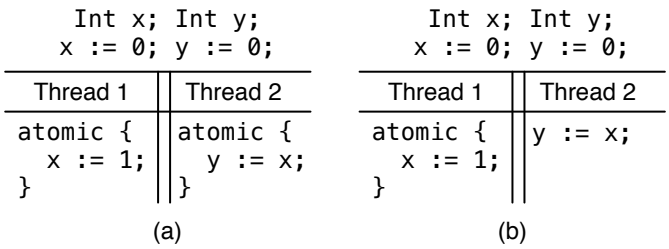


Figure 2.9: (a) Upon execution of the program the following assertion holds for the final values for x and y: $x = 1 \wedge (y = 0 \vee y = 1)$. The assertion that models the final values for (b) is $x = 1 \wedge (y = 0 \vee y = 1 \vee y = ?)$.

accesses, but also that they are isolated with non-transactional accesses. Under strong isolation the final value of y in program (b) of Figure 2.9 will be either 0 or 1. This extra level of isolation at present is too expensive to emulate efficiently in software [Abadi et al., 2009]. Chip manufacturers have shown some interest in HTM, a setting where strong isolation is efficient, but a recent attempt to bring such hardware to market was cancelled [Chaudhry et al., 2009].

2.2.2.3 Conflict Granularity

There are several types of conflict granularity in STM, the most popular being *object* [Fraser and Harris, 2007] and *address* [Harris et al., 2010]. Under object granularity all instance data is used to determine conflict detection; by contrast, address-based conflict detect treats accesses to each field distinctly. Figure 2.10 compares object and address based conflict detection.

Conflict detection occurs at a stage known as *validation*. The validation process is driven by the contention manager (discussed in Section 2.2.2.6). Put simply, validation entails asking the question "Have the accesses performed by another active or recently run transaction invalidated my view of the world?" If

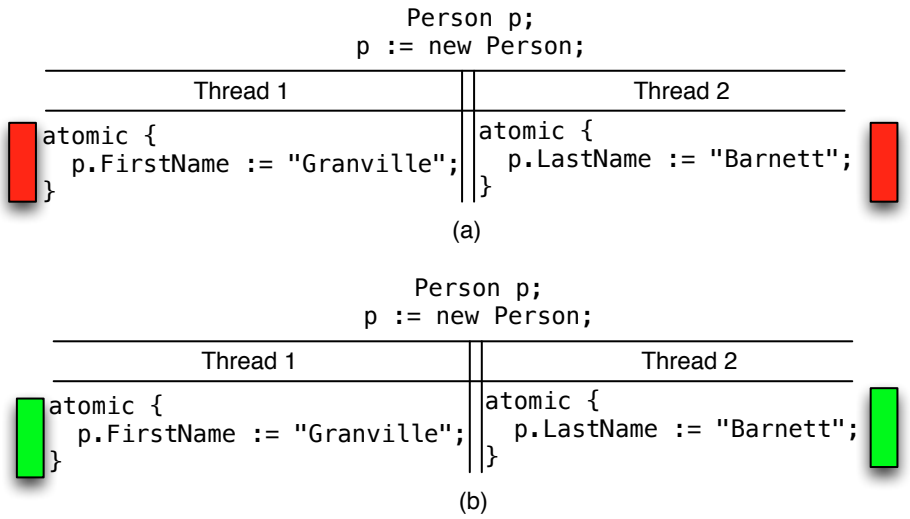


Figure 2.10: (a) Under an object STM the accesses to `FirstName` and `LastName` result in a conflict as they are both fields of the same object. (b) An address-based STM treats the accesses to `FirstName` and `LastName` distinctly as they occupy distinct regions of memory.

the answer is yes then the contention manager will select one of the conflicting transactions to commit and select the rest to abort. Validation can occur at several stages: *pre-commit* or *incremental* [Harris et al., 2010]. Pre-commit entails validating the accesses of a transaction just before it commits. By contrast, incremental validation can occur at any time during a transaction's execution. For example, in Figure 2.11 (a) uses pre-commit validation, by contrast to (b) that uses incremental validation. Incremental validation may detect memory contention earlier and therefore prevent a so-called *doomed* (a transaction that will be aborted) transaction from carrying out any further work, as such work is surplus. The poll rate of incremental validation is subject to the STM and in many respects is analogous to a thread and task scheduler. That is, the poll rate is based on a domain-specific heuristic.

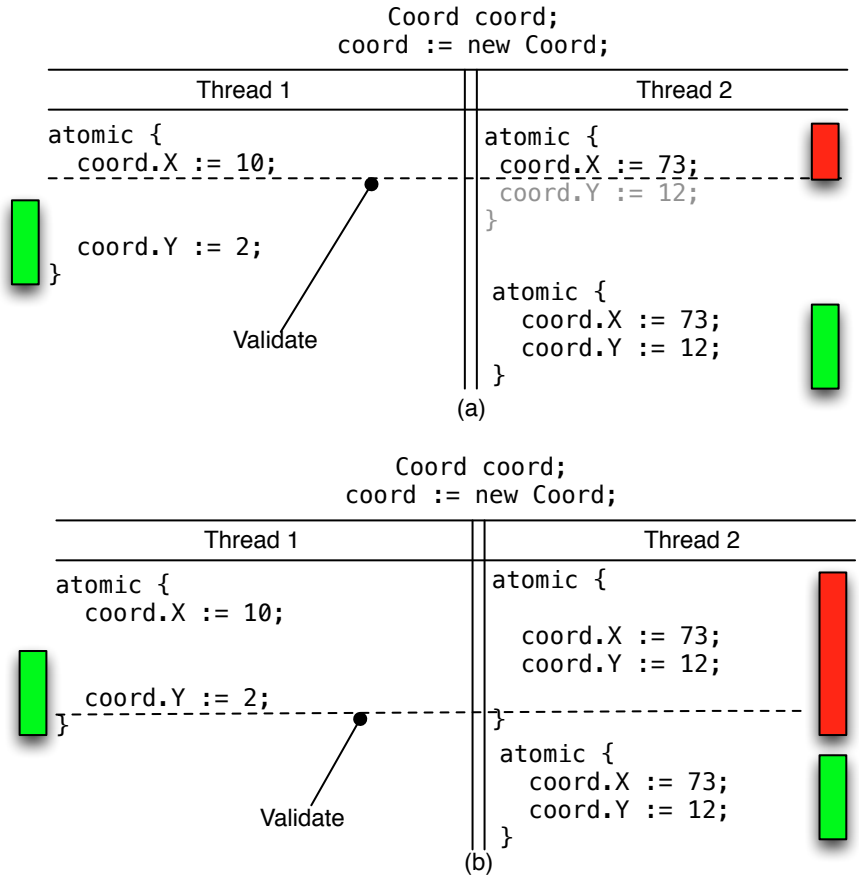


Figure 2.11: (a) Employs incremental validation at per-transactional command granularity. Thread 2's transaction is selected to abort. Here, thread 2's transaction does not execute the doomed write of `Y`. (b) Uses pre-commit validation. The conflict during the transactional execution of the accesses to `coord` are only observed upon pre-commit.

2.2.2.4 Update Mode

There are two popular types of update mode: *out-of-place* and *in-place*. We will discuss *out-of-place* as its semantics are easier to model and generally speaking is the more prevalent. Note that these two update modes may also be referred to as *indirect* and *eager version management*, or respectively *direct* and *lazy version management*. The best coverage of in-place and out-of-place update can be found in [Harris et al., 2010]. [Moore et al., 2006] present a version of in-place update that is cache friendly.

Figure 2.12 shows out-of-place update in practice. The initial value of `x = 0` and when thread 1 and 2's transactions are entered, as they each access `x`, they make a copy of `x`'s current value. Thread 1's transaction writes 1 to `x`, but the write is issued to its private copy of `x` (the transaction's redo log). The read of `x` in thread 1's transaction observes the value of `x` in thread 1's redo log. Thread 2 is similar to thread 1: it too makes a copy of `x`'s current value in memory upon issuing an access to `x`, and subsequently writes over that value upon completing its assignment of 2 to `x`. Thread 1 and 2's transactions abort so only one may commit: thread 1's transaction is selected to commit; consequently, thread 2's transaction will abort. Committing thread 1's transaction entails copying the updated values for `x` and `y` to memory. The redo log of thread 2's transaction is discarded. Upon thread 2's transaction being re-executed it will observe `x = 1` for its initial value of `x`. Thread 2's transaction subsequently commits and copies its updated value for `x` in its redo log to memory.

A distinction exists between commit and copy in out-of-place update STMs. When a transaction commits it is said to be *logically committing*. A logical

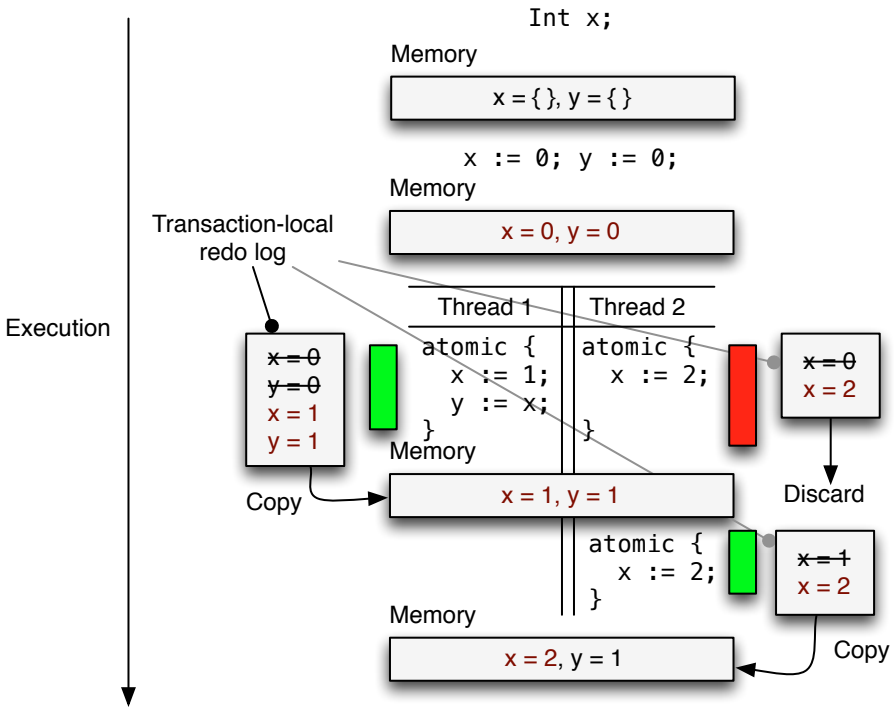


Figure 2.12: Out-of-place update. Each transaction maintains a private *redo log*. The redo log encapsulates the effect of a transaction. A transaction that commits *replays* its redo log to main memory. After this so-called replay the effect of a committed transaction is observable by the other threads of the program. Aborting transactions discard their redo logs.

commit means that the transaction has been selected to commit but its effect is not yet observable by the other active threads. A *physical commit* follows the logical commit: this is when the effect of the transaction has been propagated to memory and is observable by the other active threads in the program.

2.2.2.5 Nesting

The semantics of a transaction *nested* within another transaction vary according to STM. The three most popular semantics for nested transactions are: *flattened*; *closed*; and *open*. At the time of writing it is still an open question as to which

nesting semantics is preferable.

Flattened The simplest way to deal with nested transactions is to flatten them. The benefit of flattening is that its semantics are simple. For example, under a flattening semantics `atomic { c1; atomic { c2 } }` becomes `atomic { c1; c2 }`.

Closed The effect of a nested transaction in a closed semantics is only observable when its parent transaction commits. For example, in `atomic { x := 1 atomic { y := 1 } }`, `y == 1` is only observed if the parent transaction commits. The nested transaction may abort and not abort its parent transaction.

Open The effect of a nested transaction can persist even if its parent transaction aborts. For example, in `atomic { x := 1; atomic { y := 1 } }`, if the child transaction commits and the parent transaction aborts then `y == 1` is observed. In open nesting a committed transaction, irrespective of its nesting, has its effect being immediately observable by all active transactions. For example, in the previous example `y == 1` is observable by all transactions, not just its parent transaction, upon its commit.

For reference we summarise the most frequent semantics of each transactional property in Table 2.2.2.5. The semantics exhibited by an STM is a product of the values selected for these properties. No consensus on a standard set of semantic values exists.

Property	Semantics
Concurrency control	Optimistic or pessimistic
Update mode	In-place or out-of-place
Isolation	Weak or strong
Conflict resolution	Word-based or object-based
Validation	Incremental or pre-commit
Contention management	Heuristic driven

Table 2.2: Common semantics for transactional properties.

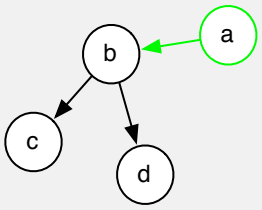
2.2.2.6 Contention Management

The contention manager is analogous to the thread or task scheduler (see Sections 2.1.1.1 and 2.1.1.2) in that it applies some heuristic to actively executing transactions to attain a specific goal, e.g. throughput or reducing the amount of wasted CPU time. Based upon the used heuristic the contention manager determines which transactions abort and commit. [Herlihy et al., 2003; Scherer and Scott, 2005; Spear et al., 2009] are largely considered the authoritative works on contention management scheduling heuristics. In this thesis contention management is treated as an oracle component that selects a transaction (randomly) to commit should several transactions conflict.

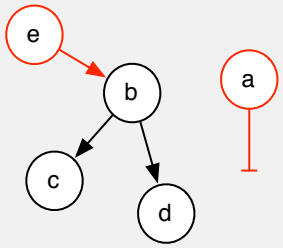
2.2.2.7 Privatisation and Publication

The privatisation and publication idioms [Spear et al., 2007] are used to permit weakly isolated STMs to execute irreversible or compute bound operations. Their application is error prone, like locks, as the programmer is required to explicitly maintain isolation invariants, but in a slightly different way than what we are used to with locks. The focus of Chapter 6 addresses the issues of applying the privatisation/publication idioms, so we now describe their facility.

```
// b and its subgraph can be reached by
// multiple threads through a
```




```
atomic {
  // Privatise.
  // cut off connectivity to
  // b by other threads
  // introduce thread-local connection
  // to b's subgraph
}
```




```
// operate on b's subgraph; no coordination required
ComplexOperation(b);
```

```
atomic {
  // Publicise.
  // make b reachable again
}
```

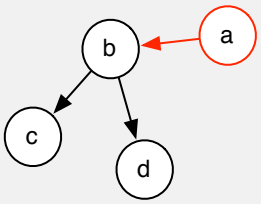


Figure 2.13: Privatising and publicising `b` and its subgraph using transactions.

With locks we use mutexes and other primitives to encode isolation semantics, e.g. in `sync(v){c;}` we are stating that the accesses issued by other threads to those which the command `c` accesses will be isolated if the other threads protect their accesses on the mutex `v`. Privatisation and publication achieves a similar type of isolation encoding via explicitly managing the reachability of a program's object graph. Modification of reachability is performed by transactions. The general thesis of the privatisation and publication idiom is shown in Figure 2.13.

Here, we wish to execute the CPU bound operation `ComplexOperation` which accesses `b` and the objects that `b` can reach. Reachable objects of `b` are located in `b`'s subgraph within the program's object graph. The first step is to private `b` by removing reachability of `a` to `b`. We state that `a` is the object which is accessible by all threads, so removing `a`'s connection to `b` prevents other threads from accessing `b`. The connection to `b` from `a` is removed using a transaction as we need to mediate the update of `a`. This is the privatisation stage. Upon its completion only the privatising thread may access `b` and its subgraph. Due to this we can perform our CPU bound operation without needing to use any form of coordination. The benefits of this in a purely transactional world are that the accesses issued to `b` and its subgraph are not transactionally instrumented as well as removing the possibility of abortion. Upon completion of our complex operation we publicise `b` and its subgraph once more by re-establishing the edge from `a` to `b`. Publication results in any mutations `ComplexOperation` made to `b` and its subgraph being observable by all other threads.

Figure 2.13 presents a simple example of applying privatisation and publication. In the author's opinion the correct application of privatisation/publication is more complex than that of correctly applying locks. The reason being that managing the connectivity of objects which are constituents of a complex object graph is very hard to do correctly. Nonetheless, privatisation/publication are powerful idioms for executing irreversible and CPU bound operations in a purely transactional setting. The current literature has attempted to address cleaner and safer semantics for the privatisation and publication idioms, which we discuss now.

[Ziarek et al., 2008] present a dynamic approach for selecting a stronger se-

mantics when a transaction attempted to execute an operation which seems (determined by a magic analysis) to require stronger guarantees than that afforded by a transaction. Unfortunately, such a semantics reverts to using programmer specified lock invariants which are error prone. [Smaragdakis et al., 2007] presented a set of language extensions to temporarily "suspend" an transaction's isolation in order to support irreversible operations, however they rely heavily on the specification of isolation invariants, which are again, error prone. Privatisation and publication [Spear et al., 2007] can be used to emulate a stronger semantics within STM but requires the programmer to correctly manage the reachability of a program's object graph. *obstinate transactions* [Ni et al., 2008] afford a strong semantics but are a product of a prior abort. [Welc et al., 2008] use single owner read locks to transition to a stronger transactional semantics but permit only a single such transaction to run at any given time. [Sonmez et al., 2009] present a model built on Haskell STM that turns transactions that access "hot" regions of memory into pessimistic transactions, however this approach again is dynamic and does not provide dataset guarantees. Autolocker [McCloskey et al., 2006] presents a model of pessimistic transactions by using a type system that uses programmer specified lock protection annotations to convert transactions into lock-based equivalents statically.

2.3 Memory Consistency Models

A *memory consistency model* defines the set of values a read may observe. All systems that admit multi-threaded programs should define a memory consistency model. All major programming platforms provide a memory consistancy model,

including the JVM, Common Language Runtime, Google Go and C++11. Surprisingly, most texts that cover these environments omit any information on their respective memory consistency model, despite it being a key factor of a concurrent program's execution semantics. In this section we discuss three memory models: program order, sequential consistency [Lamport, 1979] and the Java memory model [Manson et al., 2005]. Hardware memory models are closely related to these memory models but are not relevant to the work we present in this thesis. The best reference on hardware memory models is the tutorial by [Adve and Gharachorloo, 1996]. Memory consistency models are closely related to the work we present in Chapter 5 when we wish to relate accesses issued by transactions and locks in a simple and intuitive manner.

2.3.1 Program Order

The simplest memory model is that of *program order* (PO). The semantics of PO are restrictive but present a good starting point. Consider Figure 2.14 which executes a number of arbitrary commands. Observe that this program does not entail multiple threads of execution. That is, Figure 2.14 is single threaded. PO states that each command in Figure 2.14 appears to execute in the same order that the programmer issued them. For example, `c1` will take effect before `c2` which will take effect before `c3`, and so on. PO is a total ordering over a sequence of commands, which we represent with the binary relation $\xrightarrow{po}$. The ordering of the commands in Figure 2.14 can then be described by $c1 \xrightarrow{po} c2 \xrightarrow{po} c3$. PO seems trivial but it provides important observational guarantees to the programmer. That is, the read of `l` issued by `c1` observes the value that `c1` writes to `l`, and

so on. A memory model restricts the optimisations a compiler may perform. For example, as `c2` reads memory that `c1` writes, the compiler may not re-order the accesses issued by `c2` before `c1`. This is intuitive to the programmer as he or she wishes that, at least semantically, their program executes in the order described in their program text. This guarantee has a profound effect for memory models which govern the observational guarantees of multi-threaded programs.

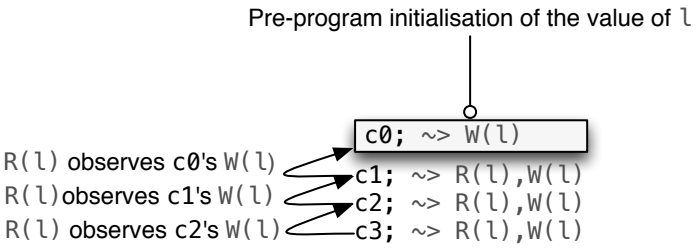


Figure 2.14: Program Order. `R` and `W` are used to denote read and respectively write. For example, `R(l)` indicates a read of `l`. Each command issues a sequence of reads and writes upon its execution. `c1`'s read observes the write of `l` by `c0`, `c2`'s read observes the write by `c1`, and so on.

2.3.2 Sequential Consistency

Program order assumes a program comprises a single thread of execution. A program that exploits a CMP is multi-threaded. Therefore, the order that each thread's instructions appear to execute to one another must be defined; otherwise, the programmer has no way to reason about the observations their code may witness. *Sequential consistancy* (SC) [Lamport, 1979] is the simplest and most restrictive memory model that governs observation guarantees for a multi-threaded program. SC states that a global total order exists, $\xrightarrow{sc}$, over the instructions executed by each thread. In $\xrightarrow{sc}$ the instructions issued per each thread do not

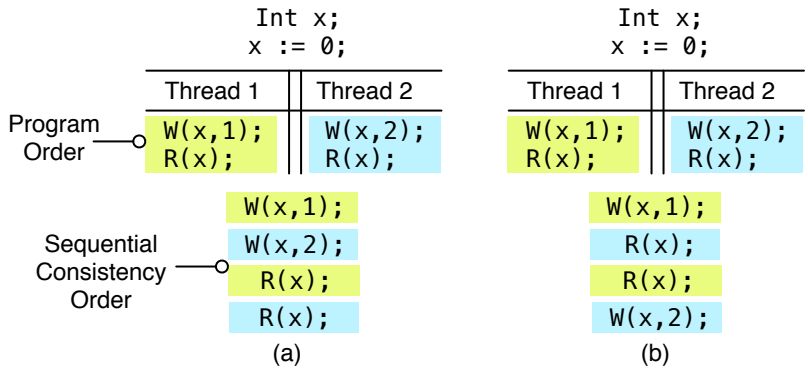


Figure 2.15: Thread 1's instructions are coloured green; thread 2's blue. `W(x,1)` writes 1 to `x`. For thread 1 we have `W(x,1)` $\xrightarrow{po}$ `R(x)` and for thread 2 `W(x,2)` $\xrightarrow{po}$ `R(x)`. (a) is valid under SC as `W(x,1)` $\xrightarrow{sc}$ `R(x)` $\xrightarrow{sc}$ `W(x,2)` $\xrightarrow{sc}$ `R(x)` preserves each thread's $\xrightarrow{po}$. By contrast, (b) does not as thread 2's read of `x` occurs before its write of `x`, which goes against the ordering of these two instructions in thread 2's PO.

invalidate their issuing thread's PO. A read within $\xrightarrow{sc}$ observes the value of the most recent write before it. Figure 2.15 shows an example of SC. Here, (a) is valid ordering under SC as each instruction that appears in $\xrightarrow{sc}$ respects its respective thread's PO. By contrast, (b) is not a valid ordering under SC as thread 2's read of `x` in $\xrightarrow{sc}$ is ordered before its write of `x`, violating thread 2's PO.

2.3.3 Java Memory Model

The Java memory model (JMM) [Manson et al., 2005] guarantees SC semantics for a correctly coordinated program. It also defines a number of orderings which help determine when the instructions executed by locks and upon `volatile` data appear to take effect. These orderings include: *synchronises-with* – a partial ordering over release and acquire instructions; *synchronisation-order* – a total order over release and acquire instructions derived from a program's execution;

and *happens-before* – the transitive closure of PO and synchronises-with order. A data race exists on a memory location x if two accesses are issued to x by distinct threads, one of them is a write and they are not ordered by happens-before. A program is correctly synchronised if all SC executions of a program are free of data races.

In Figure 2.16 (a) thread 1 writes `x` and thread 2 reads `x`. The scheduling given in (b) shows thread 1's write of `x` occurs before thread 2's. Under the JMM this scheduling is DRF, as we now explain. The JMM states that each release of `x` synchronises-with subsequent acquires of `x`. Taking Figure 2.16 (b), before thread 1 acquires the lock associated with `x` there is an initial release of `x`, otherwise `x` is not acquirable. This conceptual release synchronises-with thread 1 and 2's acquires of `x`; likewise, thread 1's release of `x` synchronises-with thread 2's acquire of `x`, and thread 2's release of `x` synchronises-with thread 1's acquire of `x`. The JMM states that a schedule of a program is DRF if the accesses to `x` are ordered by happens-before, which they are: thread 1's write of `x` takes place before thread 2's as in Figure 2.16, in which case thread 2's read of `x` is guaranteed to observe the value 1 for `x` and additionally 1 for `y`. Figure 2.16 (a) is trivially DRF as all SC executions are free of data races. The remaining semantics that the JMM defines is to protect the strong security and safety guarantees of the JVM, see [Manson et al., 2005] for more details.

2.4 Summary

There are three general elements which aid in the successful reasoning of a concurrent program: the language abstractions provided by the host programming

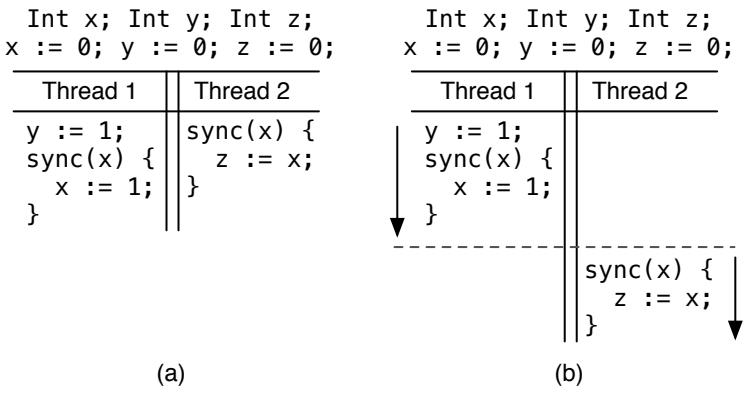


Figure 2.16: (a) Thread 1 writes `x` and thread 2 reads `x`. (b) a DRF scheduling of (a) according to the JMM. Here, thread 1 and 2's accesses of `x` are ordered by happens-before.

language and its associated libraries and runtime environment; static and dynamic tools which aid the programmer in detecting concurrency related errors in their programs; and the semantics afforded by the host's memory consistency model. Together they provide a compelling programming model for designing correct concurrent programs. We use the term correct in this thesis as a synonym for data-race-freedom, although the term can more broadly encapsulate other criteria such as deadlock freedom, as well as others. The remainder of the thesis presents innovations that touch on each of the aforementioned categories: Chapter 5 presents an abstract memory consistency model for programs that use both locks and transactions to coordinate accesses to shared memory; Chapter 6 gives a programming language construct for simplifying the application of the privatisation and publication idioms; and Part II presents a static analysis for automatically determining the data-race-freedom of programs that use both locks and transactions to coordinate accesses to shared memory.

Part I

Dynamic Reasoning

In this part of the thesis we present two novel techniques to dynamically reason about the semantics of a concurrent program: *moverness* and *guaranteed transactions*.

Chapter 3: We introduce the role that reads and writes play in determining the observable semantics of concurrent programs. We then describe means to serialise them using locks and transactions, and the situations in which each is the appropriate tool. The chapter concludes by giving illustrattive examples of situations when each tool excels, giving an intuition of why a programmer may wish to use both in their program.

Chapter 4: Gives the programming model that the subsequent chapters in the dynamic reasoning part of the thesis are based upon. Locks and transactions are used to serialise accesses to shared memory. We then define the semantics of locks and transactions via a small step operational semantics.

Chapter 5: We reason about the direction which reads and writes issued by locks and transactions may travel in upon instances of memory contention. We describe the set of permissible directions by defining *moverness*. Locks are found to be *left movers* due to their non-abortable semantics and transactions *right movers* as they may be aborted. Non-conflicting locks and transactions are *free movers*, and transactions and locks with respect to themselves are *both movers*.

Chapter 6: An alternative to locks for certain scenarios is presented in the form of *guaranteed transactions*. A guaranteed transaction affords pessimistic serialisation but without the programmer having to explicitly manage isolation invariants or the reachability of the object graph. A key benefit of

guaranteed transactions is their abstract parity with transactions. We also define their moverness with respect to transactions, and find them to be left movers. Guaranteed transactions can be considered a half way house between transaction and lock semantics.

Chapter 3

Introduction

The observable semantics of a multi-threaded program are a consequence of the control flows taken by each thread and the interleaving of each thread's issued accesses. In this chapter we give an overview of how reads and writes affect program semantics. We also discuss how the effect of reads and writes can be strictly defined by using locks and transactions to serialise their execution.

3.1 Actions

Understanding the semantics of an executing program is seldom trivial, particularly for concurrent programs. Reasoning about the semantics of a concurrent program requires the programmer understand when *actions* (reads, writes, among other operations) issued by distinct threads *may* take place simultaneously. The possible permutations in which these actions take affect determines the observable values yielded by the execution of a multi-threaded program. For example, Figure 3.1 shows a program where two threads write to `y`. Here, there are three

possible schedules that influence the final value observed for `y`: thread 1 writes `y`, followed by thread 2's write, or vice versa; or, thread 1 and thread 2's write of `y` take place concurrently. For the first two cases the final value obserbed for `y` is most likely that we expected. However, in the latter case we may observe a value for `y` that is neither 1 or 2. In this instance we observe a value of `y` that is a consequence of a data race (Section 2.2), which are, unfortunately, common in multi-threaded programs. Preventing data races is the topic of Part II.

```
Int y;
y := 0;
```

Thread 1	Thread 2
`y := 1;`	`y := 2;`

Figure 3.1: Threads 1 and 2 write `y` but their writes may overlap in time, resulting in a data race.

3.2 Action Indivisibility

Locks and transactions can be used to restrict the ability of threads to concurrently issue accesses to defined regions of memory. We state this facility as the ability of a thread to *serialise* its accesses with respect to those issued by other active threads. Provided the programmer applies lock and transactional semantics correctly, he can expect to observe data values that are a consequence of a well-defined permutation of actions. The programmer can do this due to a combination of two semantics: first, that of locks and transactions; and secondly that of the underlying memory model (Section 2.3). We will briefly look at lock and transactional semantics now and defer a discussion of memory models to Chapter 4. We refer the reader to Chapter 2 for more information.

3.2.1 Locks

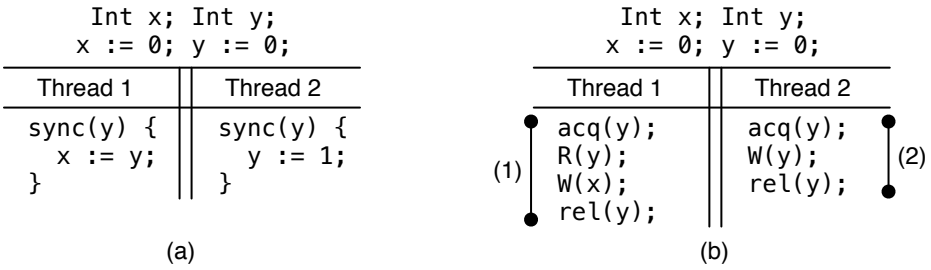


Figure 3.2: (a) Each thread's access of `y` is protected by the same mutex. Consequently, each thread's access of `y` is isolated. (b) Shows the coversion of (a) to its synchronisation and read/write action form. Due to each thread's access of `t` being isolated the acquire/release delimited sequence of actions collapses into a single indivisible action. For example, if we label (1) as action a_1 and (2) as action a_2, the possible execution sequences are a_1a_2 or a_2a_1.

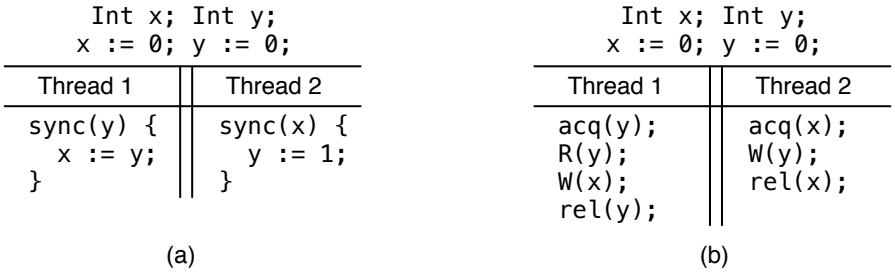


Figure 3.3: (a) Each thread uses a different mutex to protect its access of `y`. Consequently, each thread's access of `y` is not isolated. (b) Due to the locks not agreeing on a mutex each thread's acquire/release delimited sequence of actions is not treated as an indivisible action. Therefore, the possible action sequence is any permutation of the four actions issued by thread 1 and the three actions issued by thread 2.

Lock issued accesses to the same memory by distinct threads are treated as being indivisible if the locks are protected on the same mutex. For example, consider the program in Figure 3.2. Here, locks are used to protect each thread's accesses. The use of each lock constructs a sequence of actions delimited by the

synchronisation actions acquire and release. Each thread's lock issued access of `y` is isolated with respect to the other thread's access of `y` as both locks use the same mutex. Because the accesses of `y` are isolated they will be serialised. That is, there are only two possible schedules for Figure 3.2: either thread 1's read of `y` takes effect, then thread 2's write of `y`, or vice versa. Due to each thread's access of `y` being isolated we can treat the sequence of constituent accesses issued by each thread's lock as if it were a single indivisible action. By contrast, in Figure 3.3 each thread's sequence of actions may not be treated as an indivisibl action as the accesses of `y` are protected by different mutexes.

3.2.2 Transactions

Indivisibility of transactionally issued accesses is not guaranteed. This is particularly the case for transactions in a weakly isolated STM (Section 2.2.2.2), which are the semantics of the STM we use throughout the thesis. The key concept in a weakly isolated STM is that transactional accesses are isolated with respect to other transactional accesses. For example, the accesses of `y` in Figure 3.4 are isolated but those in Figure 3.5 are not. If the accesses issued by a transaction are isolated then we can treat the sequence of actions the transaction issues as a single indivisible action, like in Figure 3.4. By contrast, transactional accesses that are not isolated cannot be treated as an indivisible action, as shown in Figure 3.5.

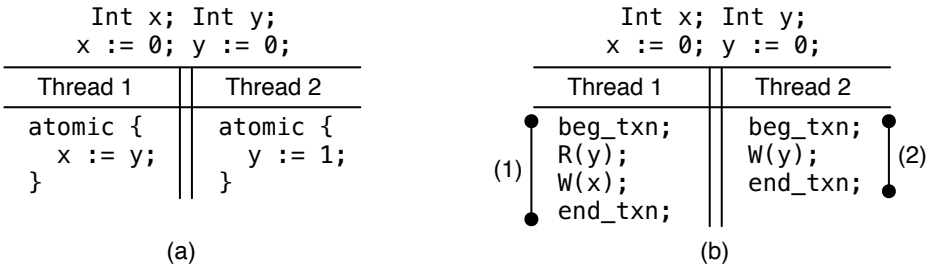


Figure 3.4: (a) Each thread's access of `y` is isolated as their respective accesses are issued transactionally. (b) Each transaction begin/end delimited sequence of actions can be treated as an indivisible action. For example, if we label (1) as the action a_1 and (2) as the action a_2, the sequences a_1a_2 or a_2a_1 are possible.

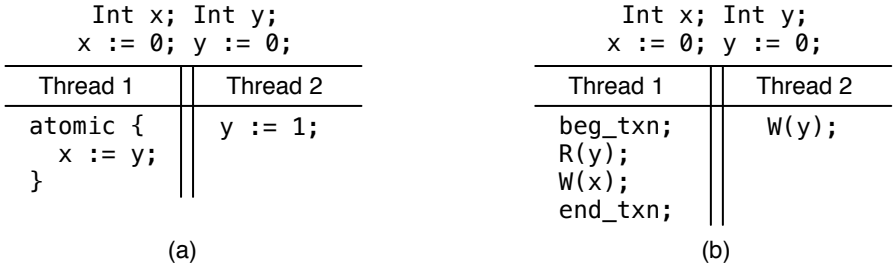


Figure 3.5: Accesses of `y` are not isolated. The uncoordinated access of `y` by thread 2 results in thread 1's transactional sequence of actions not being viewed as taking effect indivisibly.

3.3 Locks *or* Transactions

3.3.1 Locks

Locks have been the mainstay for facilitating serialisation in multi-threaded programs for decades. Virtually all thread safe libraries use locks to some extent. The designers of modern languages such as Java and C# felt that locks were so important that they made them a fundamental part of the respective languages. By contrast, C and C++, prior to C11 and C++11, have relied on libraries such as pthreads [Butenhof, 1997] to provide their concurrency semantics. Two points

```
Int x; Int y;
x := 0; y := 0;
```

Thread 1	Thread 2

`sync(x) {`
`  x := y;`
`}`
	`sync(y) {`
	`  y := 1;`
	`}`

Figure 3.6: Threads 1 and 2 access `y`. However, each thread's access of `y` is protected by a different mutex. Therefore, thread 1's read and thread 2's write of `y` may take place concurrently and result in a data race.

of friction are common when applying locks: explicit invariant management and composition.

Explicitly managing invariants is often error prone. As an analogy we can consider the maintenance of lock isolation invariants to being akin to manual memory management. That is, while the concept is often trivial to grasp, its application in practice is easy to get wrong. Unfortunately, the incorrect maintainence of lock invariants can lead to complex program errors such as data races and deadlock. Figure 3.6 gives an example of a program that leads to a data race.

A second problem with locks is that of composition. Modern software design is based upon the concept of resuable components. For example, one company may provide a library A and another library B. A programmer would like to use A and B as each provides complimentary functionality. For a single threaded program we can compose A and B in an often intuitive manner. However, in a multi-threaded program it is possible that A and B mutate data which may be accessible by several threads. Consequently, the programmer must serially compose A and B. Using locks this is non-trivial as it requires the programmer to compose lock invariants. Figure 3.7 gives an example of such a lock invariant

```
ComponentA a;
ComponentB b;
```

Thread 1	Thread 2

`sync(a) {`
`  sync(b) {`
`  a.apply(b);`
`  }`
`}`
	`sync(b) {`
	`  sync(a) {`
	`  a.apply(b);`
	`  }`
	`}`

Figure 3.7: Threads 1 and 2 compose the components `a` and `b`. Because `a` and `b` can be accessed by multiple threads we pessimistically compose them with locks. The programmer working on the program text executed by thread 1 composes the isolation invariants in the sequence of acquiring `a` then `b`; the programmer who coded the program text being executed by thread 2 took the opposite approach. The result is deadlock should thread 1 acquire `a` and thread 2 acquire `b`.

composition. The more components the programmer wishes to compose, the harder it becomes to compose isolation invariants and still maintain the desired serialisation semantics.

3.3.2 Transactions

STM is an *alternative* to locks for mediating accesses to shared memory. The semantics afforded by transactions are often too weak for operations which are irreversible or demand run once semantics. For example, Figure 3.8 shows a program which executes a seemingly irreversible operation. Here, transactions are a bad choice as the operation being performed cannot be reversed. That is, should the transaction abort it is likely that the atomicity and consistancy guarantees of STM will be violated. CPU bound operations, such as that shown in Figure 3.9, are impractical to be executed transactionally. Here, the problem is that an operation, despite the fact it may have utilised several seconds of CPU time, may be aborted introducing contention on system resources.

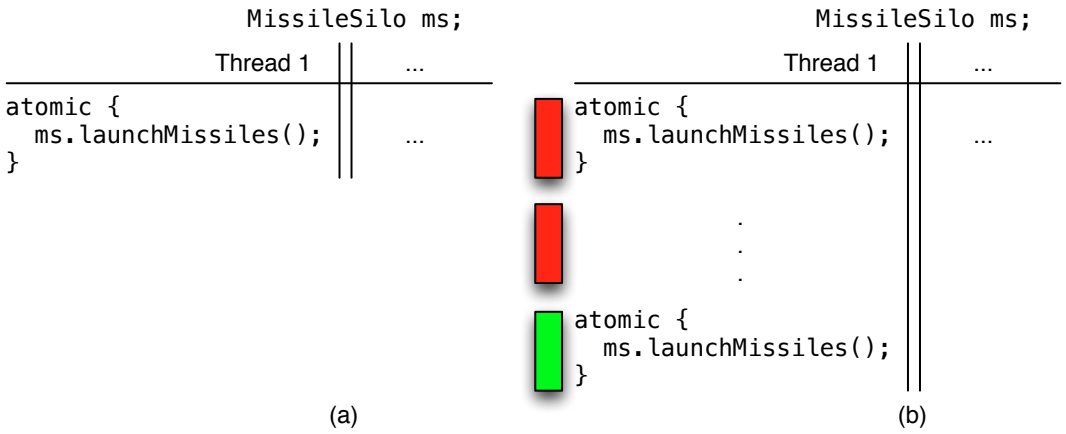


Figure 3.8: (a) Thread 1 launches some missiles. Once the missiles are launched it may not be possible to have them aborted, e.g. the missles may be out of control range. This problem is exemplified in (b) where the transaction executing `launchMissiles` is aborted several times before it finally commits.

3.4 Locks *and* Transactions

Most multithreaded libraries written in a language like Java use locks extensively. Transactions must co-exist with locks in the same program, otherwise the attraction of languages such as Java – its libraries – are of little use. In this section we will discuss how locks and transactions can be used to compliment one another by describing their respective strengths. Generally speaking, locks facilitate low friction strong serialisation semantics, while transactions reduce the complexity of correctly serialising component composition.

Consider Figure 3.10 where transactions are used to write data to disk. Here, transactions may lead to data inconsistancy on the disk should a transaction abort. Transactions are not appropriate for executing such operations, but locks are, as shown in Figure 3.11. Locks are also appropriate for executing CPU bound operations as shown in Figure 3.12. Using locks for executing such operations

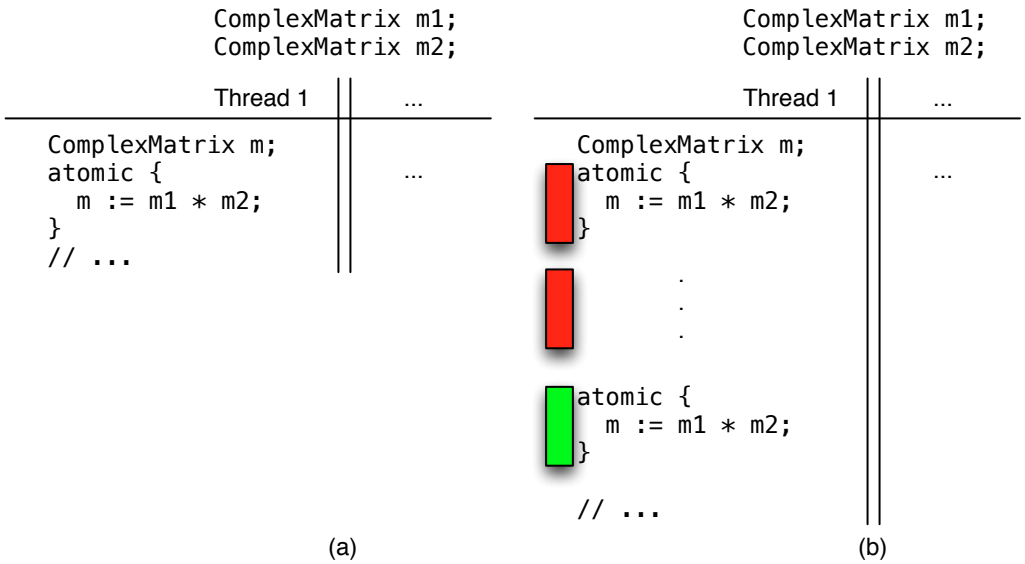


Figure 3.9: (a) Shows a program that performs the CPU bound operation of multiplying two complex matrices. In (b) the transaction executing the matrix operation is aborted several times before committing. Here, an operation which may have taken at most 100 milliseconds of CPU time ends up taking several seconds, introducing artificial contention on system resources.

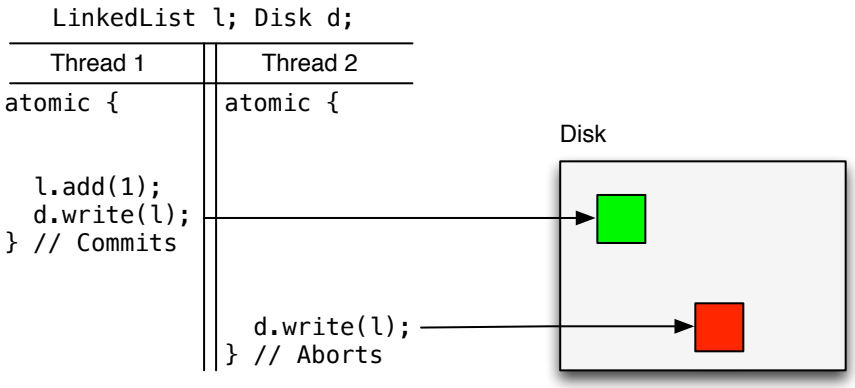


Figure 3.10: Using transactions to execute an irreversible I/O operation. Thread 2's transaction aborts but its write to disk remains.

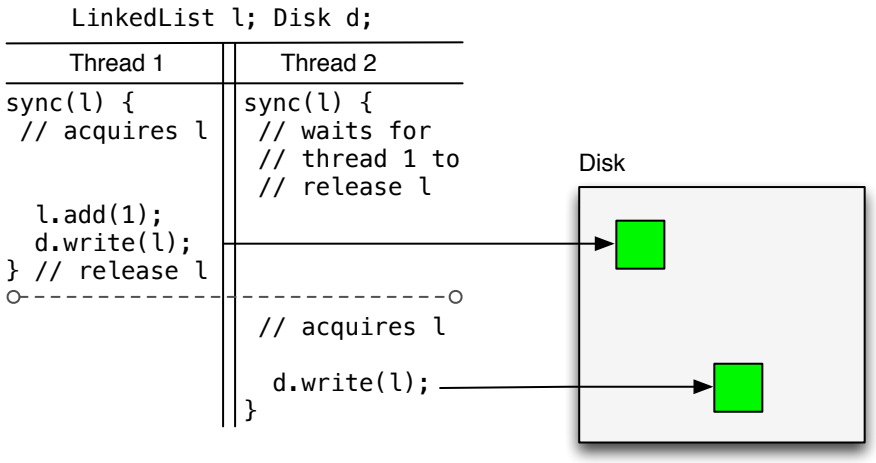


Figure 3.11: Using locks to safely execute an irreversible I/O operation.

comes at the expense of the programmer having to maintain isolation invariants. Other approaches are possible, for example we might introduce a mutex which is to be acquired before we access `m1` and `m2`, as shown in Figure 3.13 (a). Another key strength of locks is that they can be directly influenced by the programmer. For example, the programmer may explicilty partition read and write cases as shown in Figure 3.13 (b).

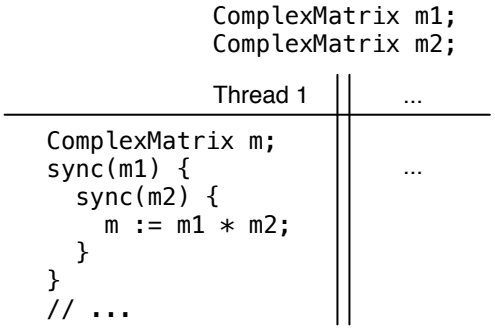


Figure 3.12: Locks are used to execute a CPU bound operation.

The semantics of transactions are not as easily influenced as locks. The reason

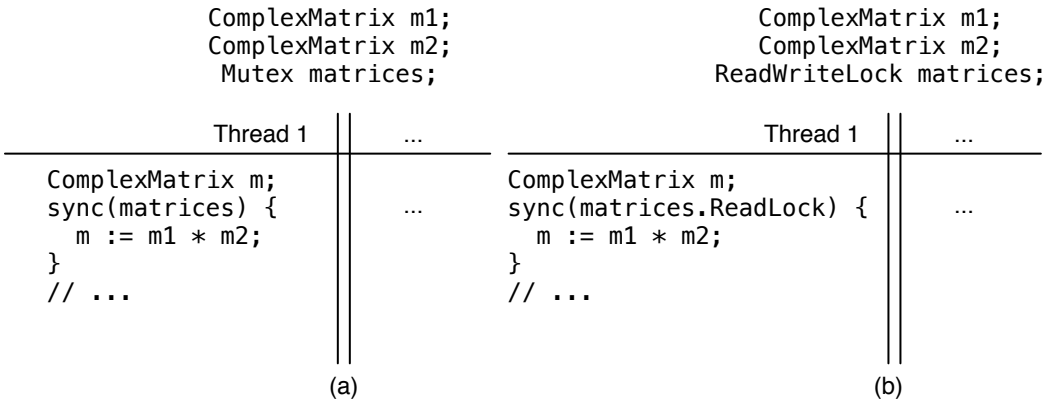


Figure 3.13: (a) The programmer defines the object `matrices` which is to be used each time an operation accesses the matrices `m1` and `m2`. The lock invariant is simplified at the cost of increasing the granularity of the isolation invariant. (b) A Read/Write lock is used to optimise for cases when `m1` and `m2` are only read. Threads that only read `m1` and `m2` need only acquire the read lock.

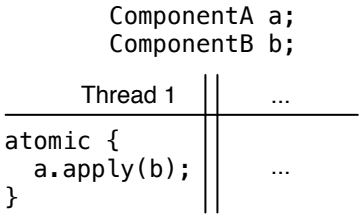


Figure 3.14: Transactions are used to simplify component composition.

is analogous to optimising memory management in a garbage collected environment such as the JVM. That is, in order to optimise memory management the programmer's actions must compliment the semantics of the underlying service. In our case the service is STM. The key feature of STM is the ease at which it can be used to compose operations without burdening the programmer with maintaining isolation invariants. Figure 3.14 shows a typical example of using transactions to compose components. Under STM the programmer seldom has to put much thought into the act of composing components.

3.5 Summary

Taken individually locks and transactions are both insufficient for effectively solving many general purpose coordination scenarios in concurrent programs. Locks can be considered the "assembly language" of coordination – they permit the construction of most mutual exclusion idioms. However, locks are hard to use, particularly when composing software components. Transactions do not provide run once semantics like locks, but they do offer a simple and intuitive composition semantics without burdening the programmer with complex isolation invariants. The use of locks and transactions in the same program permits the programmer to pick and choose the desired semantics for the task at hand: locks are ideal to execute I/O and CPU bound operations; by contrast, transactions simplify component composition and alleviate the programmer from maintaining isolation invariants.

Chapter 4

Programming Model

4.1 Programming Language

The programming language that we use is given in Figure 4.1. Most of the language features are standard with the exception of $\mathsf{atomic}\{c\}$ and $\mathsf{sync}(v)\{c\}$ which we explain shortly. A simple version of object oriented programs are permitted via the use of classes and methods. In our examples classes are generally used to structure data and determine the connectivity of a program's object graph, which is our main focus.

4.1.1 Locks

The locks supported, denoted syntactically by $\mathsf{sync}(v)\{c\}$, protect execution of the commands c according to the semantics of the mutex v. In Java this type of lock is known as an *explicit lock* [Arnold et al., 2005]. Given the parallel composition $\mathsf{sync}(v_1)\{c_1\} \parallel \mathsf{sync}(v_2)\{c_2\}$ the accesses issued by c_1 and c_2 are isolated if and only if $v_1 = v_2$. Locks can be recursively acquired/released. We clarify these

$$
\begin{array}{lll}
\text{Program} & ::= & \text{Class-Decl}^{*}\,(cn\ v)^{+}\ (v := \mathsf{new}\ cn)^{*} \\
 & & (\ \mathrm{T}\ \|\ \ldots\ \|\ \mathrm{T}\ ) \\
\text{Class-Decl} & ::= & \mathsf{class}\ cn\ \{ \\
 & & \quad (cn\ v)^{+} \\
 & & \quad \text{Meth-Decl}^{*} \\
 & & \} \\
\text{Meth-Decl} & ::= & m((cn\ p)^{*})\ \{ \\
 & & \quad \mathrm{C} \\
 & & \} \\
b \in \text{BExpr} & ::= & v \neq \mathsf{null} \mid v = \mathsf{null} \mid \mathsf{True} \mid \mathsf{False} \\
\mathrm{T} & ::= & (cn\ v)^{*}\ \mathrm{C} \\
c \in \mathrm{C} & ::= & v := x \\
 & & \mid v := x.f \\
 & & \mid v.f := x \\
 & & \mid v.m(p^{*}) \\
 & & \mid \mathsf{atomic}\{c\} \\
 & & \mid \mathsf{sync}(v)\{c\} \\
 & & \mid v := \mathsf{new}\ cn \\
 & & \mid \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} \\
 & & \mid \mathsf{while}\ b\ \{c\} \\
 & & \mid c_1;c_2
\end{array}
$$

Figure 4.1: Programming Language Abstract Syntax.

semantics, along with the isolation semantics of lock and transactional accesses in Section 4.2.

4.1.2 Transactions

Transactions are denoted syntactically by $\mathsf{atomic}\{c\}$ which states that the commands c are to be executed under a transactional semantics. Unlike locks, no one semantics for transactions are standard, so we now give the semantics of the transactions we model.

Weakly Isolated: transactional accesses are isolated only with respect to accesses issued by other transactions. (We define the isolation of transactional and lock accesses in Section 4.2.)

Conflict Granularity: transactional accesses conflict at the granularity of memory locations.

Update Mode: transactional accesses are issued out-of-place. That is, each transaction updates a local copy of its dataset, the transaction's *redo log*, which becomes observable only should the transaction commit.

Nesting: nested transactions are flattened , e.g. $\mathsf{atomic}\{c_1; \mathsf{atomic}\{c_2\}\}$ becomes

$\mathsf{atomic}\{c_1; c_2;\}$.

Each lock and transactional instance is associated with a label id, e.g. $\mathsf{atomic}\{c\}$ becomes $\mathsf{id}{:}\mathsf{atomic}\{c\}$ and $\mathsf{sync}(v)\{c\}$ becomes $\mathsf{id}{:}\mathsf{sync}(v)\{c\}$, which takes on a unique integer identifier *id* each time it is encountered within the program text. A nested lock within a transaction and vice versa is prohibited[1].

4.2 Operational Semantics

We now present the operational semantics for the language given in Section 4.1. The definitions of the functions referenced can be found in Appendix A.

[1]No clear consensus on a semantics for this situation exists. The simplest option is to use a single global lock atomicity semantics. We address a similar issue when we introduce guaranteed transactions in Chapter 6.

4.2.1 Overview

There are several pieces to our semantics so we begin with a high level overview of how the respective configurations and rules relate to one another. Figure 4.2 gives a diagrammatic overview of a program's execution. On first reading one should skim this section and then return to it after reading the rest of the chapter.

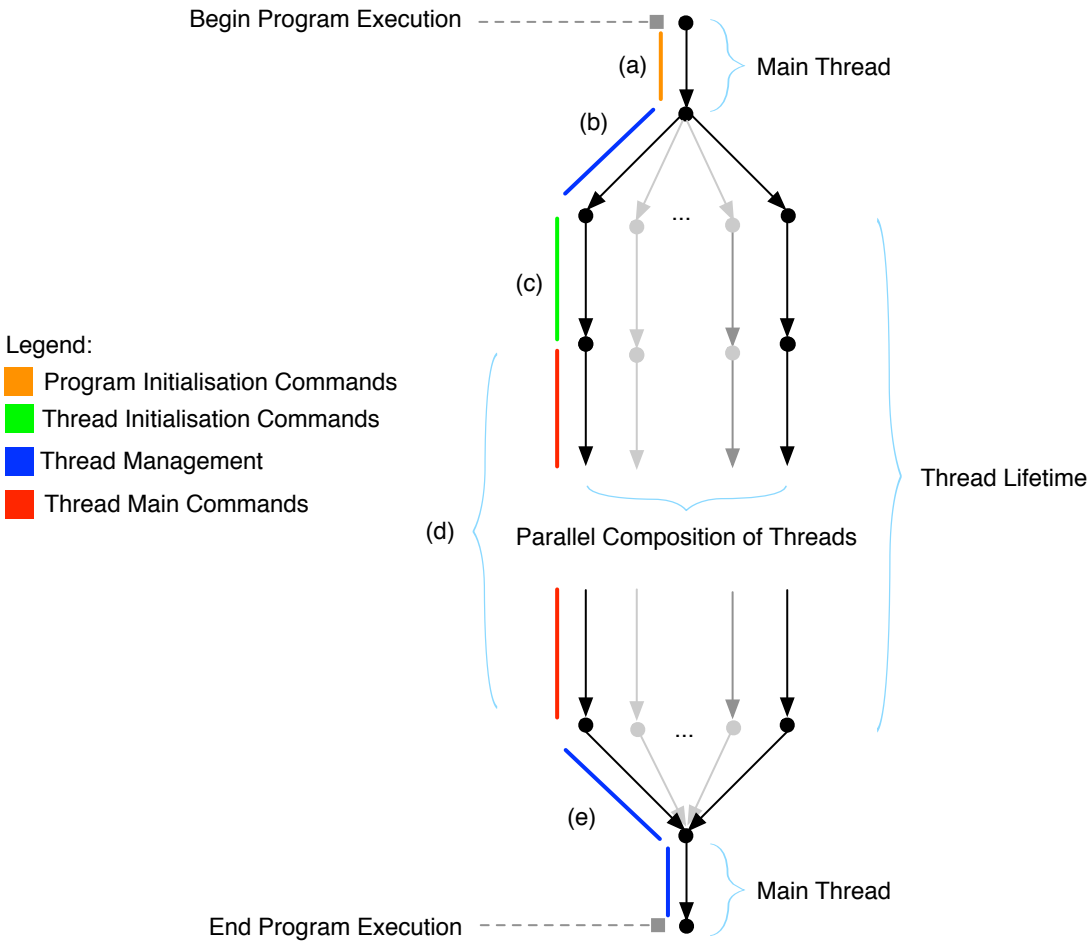


Figure 4.2: Annotated program execution lifetime.

A program's execution undergoes the following phases:

1. The main thread executes global initialisation commands ((a) in Figure 4.2), e.g. global variable declarations and allocation of objects. The relevant rules are (PROGRAM−INIT−NEW) and (PROGRAM−INIT−VAR−DECL).

2. The main thread forks several threads ((b) in Figure 4.2) which are treated as a parallel composition. In Figure 4.2 the act of forking a thread falls under the category of thread management, as does joining which we cover shortly. The rule that forks the parallel composition of threads is (PROGRAM−FORK).

3. Each thread then executes its initialisation commands ((c) in Figure 4.2) which are thread-local variable declarations. The variable declarations are executed by (THREAD−INIT−VAR−DECL).

4. Each thread executes its non-initialisation commands, (d) in Figure 4.2). Each non-initialisation command is executed by a thread under one of three coordination semantics: uncoordinated, transactional or lock-based. The rules that govern the execution of a thread's non-initialisation commands are the thread rules in Section 4.2.4.2 and the unified rules in Section 4.2.4.3.

5. When each thread has executed its non-initialisation commands a join operation is performed, (PROGRAM−JOIN), (e) in Figure 4.2. Upon the join completing the program ceases execution.

4.2.2 Configurations

On a first reading it is recommended that the reader skims this section, consults Section 4.2 and then returns should further clarification of a configuration's

components be required.

4.2.2.1 Program

A program configuration is of the form $\langle C_{pinit}, T_1 \mid\mid \ldots \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{ld} \rangle$, where:

- C_{pinit} are the program initialisation commands that are executed before the parallel composition of threads are spawned. The commands making up C_{pinit} are variable declarations, $cn\ v$, and object allocations, $v := \mathsf{new}\ cn$. See Program in Figure 4.1.
- $T_1 \mid\mid \ldots \mid\mid T_n$ is a parallel composition of thread configurations, discussed in Section 4.2.2.2. The thread configurations are formed by the rule (PROGRAM−FORK).
- $\sigma \in \mathsf{State} \stackrel{\mathsf{def}}{=} \mathsf{Store} \times \mathsf{Heap}$ represents the program state. The syntax "$\sigma \in \mathsf{State}$" asserts σ is an instance of the type State. $\mathsf{Store} \stackrel{\mathsf{def}}{=} \mathsf{Variable} \rightarrow \mathsf{Location} \times \mathsf{Location}$ maps a variable identifier to a tuple whose first component is the memory location of the variable and second component its value. Variable contains all possible contiguous sequences of the characters $a, \ldots, z$ and Location comprises all possible memory locations. We use the metavariable ℓ and its subscripts to range over memory locations. $\mathsf{Heap} \stackrel{\mathsf{def}}{=} \mathsf{Location} \rightarrow \mathsf{Object}$ maps a memory location to an object, where $\mathsf{Object} \stackrel{\mathsf{def}}{=} \mathsf{Field} \rightarrow \mathsf{Location} \times \mathsf{Location}$ maps a field identifier to a pair whose first component is the location of the field and second component its value. Field is defined similarly to Variable, e.g. `name` is both a valid instance of Field and Variable. The second component of a variable or field is null when the value of the variable or respectively field is a primitive, e.g. an integer.

- $\mathsf{fs} \in \mathsf{FS} \stackrel{\mathsf{def}}{=} \mathsf{LocationSet}$, where $\mathsf{LocationSet}$ is a set of $\mathsf{Location}$. fs represents the program's free store. That is, fs is a set which comprises the memory locations allocated by an executing program.

- $\mathsf{md} \in \mathsf{MD} \stackrel{\mathsf{def}}{=} \mathsf{ID} \rightarrow \mathsf{MetaData}$ maps a unique label $\mathsf{ID} \stackrel{\mathsf{def}}{=} \mathbb{N}$ associated with a lock or transaction instance to its respective metadata. $\mathsf{MetaData} \stackrel{\mathsf{def}}{=} \mathsf{Time} \times \mathsf{Time} \times \mathsf{LocationSet} \times \mathsf{LocationSet} \times \mathsf{LocationSet} \times \mathsf{Coord}$:

  - The first two components represent the begin and respectively commit time of the lock or transaction, where $\mathsf{Time} \stackrel{\mathsf{def}}{=} \mathbb{N}$.

  - The three components of type $\mathsf{LocationSet}$ represent the read set, write set and respectively dataset of the lock or transaction. Recall that the dataset is the union of the read and write set. We include the dataset in a lock and transaction's metadata to permit simpler construction of of our operational semantics which we give later.

  - The last component of $\mathsf{MetaData}$ denotes the type of coordination the metadata is modelling, where $\mathsf{Coord} \stackrel{\mathsf{def}}{=} \mathcal{L} \mid \mathcal{A}$. The label $\mathcal{L}$ denotes a lock and $\mathcal{A}$ a transaction. $\mathcal{L}$ is parameterised on two values: a thread identifier τ and a handle count $count$, $\mathcal{L}(\tau, count)$. These parameterised values are used to support nested and recursive locks.

- $\mathsf{Id} \in \mathsf{ID}$ holds the value which the next unique label is the successor of.

4.2.2.2 Thread

A thread configuration is $\langle \tau, C_{tinit}, C, \mathsf{s}_\tau, \delta \rangle$.

- $\tau \in \mathcal{T}$ is a unique integer representing a thread identifier. $\mathcal{T}$ is the set of active thread identifiers. For example, $\mathcal{T} = \{1, 2, 3\}$ if the program configuration comprises the parallel composition of threads $T_1 \parallel T_2 \parallel T_3$.

- C_{tinit} is the sequence of thread initialisation commands. These commands are restricted to variable declarations, see T in Figure 4.1. The variables declared by a thread's initialisation commands are accessible only by the defining thread. All the commands in C_{tinit} are executed before a thread's non-initialisation commands C.

- C is the sequence of non-initialisation commands to be executed by the thread.

- $\mathsf{s}_\tau \in \mathsf{Store}$ is the thread's local store. s_τ is defined only for the variables declared in the command sequence C_{tinit}.

- $\delta \in \mathsf{State}$ is a redo log and is only present during the execution of a transaction.

4.2.2.3 Unified

A unified configuration is $\langle \tau, c, \delta, \mathsf{fs}, \gamma_\mathsf{R}, \gamma_\mathsf{W}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld} \rangle$, where:

- τ is the active thread identifier.

- c is the command to execute.

- δ is the threaded state pair.

- fs is the program's free store.

- $\gamma_{\mathsf{R}} \stackrel{\mathsf{def}}{=} \mathsf{LocationSet}$ and $\gamma_{\mathsf{W}} \stackrel{\mathsf{def}}{=} \mathsf{LocationSet}$ are read and respectively write sets. Read and write sets are only used when executing a command transactionally; otherwise, they are set to $\perp$, "undefined."

- s_τ, σ, md and ld are the thread's local store, the global state, the metadata mapping and respectively the currently taken unique identifier label. These components are only set when executing nested locks; otherwise, they are set to $\perp$.

All commands executed under an uncoordinated, lock or transactional semantics delegate their execution to a unified configuration. The advantage of the unified configuration is that we can define the semantics of a command c once and then "thread-in" the appropriate components depending on the coordination semantics c is to be executed under.

4.2.3 Transition Relations

4.2.3.1 Program

There are two forms of reduction for a program configuration: one for when executing the initialisation commands of the main thread, and another when executing the commands of the parallel composition of threads.

Initialisation Commands Executing the initialisation commands of the main thread results in $P \xrightarrow{\lambda^+} P'$, where

$$P{=}\langle c, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{ld}\rangle \quad P'{=}\langle c', T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma', \mathsf{fs}', \mathsf{md}, \mathsf{ld}\rangle$$

Execution of an initialisation command by the main thread only ever updates the σ and fs components of the program configuration. Each reduction generates a sequence of actions λ^+ which we discuss shortly.

Non-Initialisation Commands The non-initialisation commands of a program are those executed by the parallel composition of threads that the program spawns. Executing the commands of the threads within the parallel composition results in $P \xrightarrow{\Lambda_i \;||\; \Lambda_j \;||\; \Lambda_k \;||\; \Lambda_m \;||\; \Lambda_u} P'$, where

$$P{=}\langle \epsilon, T_i \;||\; \ldots \;||\; T_j \;||\; \ldots \;||\; T_k \;||\; \ldots \;||\; T_m \;||\; \ldots \;||\; T_u, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{ld} \rangle$$
$$P'{=}\langle \epsilon, T_i \;||\; \ldots \;||\; T'_j \;||\; \ldots \;||\; T'_k \;||\; \ldots \;||\; T'_m \;||\; \ldots \;||\; T'_u, \sigma', \mathsf{fs}', \mathsf{md}', \mathsf{ld}' \rangle$$

Note that the commands of the parallel composition are only executed after the initialisation commands of the program.

- For now we assert thread T_i is executing a lock that has not acquired its mutex, T_m a transaction which is committing, T_u an uncoordinated command, T_j a lock which has acquired its mutex and T_k an aborted transaction. Threads T_j, T_m and T_u contribute to the updated program state σ'. Threads T_j, T_k and T_m contribute to md' and ld'. We cover this reduction further in Section 4.2.5.

- Upon a program reduction each thread executes a sequence of actions that conforms to one of the sequences defined by Λ, defined in Figure 4.3. The actions from each respective sequence can be executed concurrently in any order so long as they respect their issuing thread's program order.

$$
\begin{array}{lcl}
\lambda & \stackrel{\mathsf{def}}{=} & \mathsf{R} \mid \mathsf{W} \mid \mathsf{TBEG} \mid \mathsf{TABT} \mid \mathsf{TCMT} \mid \mathsf{ACQ} \mid \mathsf{REL} \mid \mathsf{NOP} \\
\lambda_{RW} & \stackrel{\mathsf{def}}{=} & \mathsf{R} \mid \mathsf{W} \\
\Lambda & \stackrel{\mathsf{def}}{=} & \lambda_{RW}^{+} \\
 & & \mid \mathsf{TBEG}\ \lambda_{RW}^{+}\ (\mathsf{TABT} \mid \mathsf{TCMT}) \\
 & & \mid \mathsf{ACQ}\ \lambda^{+}\ \mathsf{REL} \\
 & & \mid \mathsf{NOP}
\end{array}
$$

Figure 4.3: Abstract Syntax for Actions.

Figure 4.3 shows the abstract syntax of actions which are issued during a reduction:

- R is a read.
- W is a write.
- TBEG delimits the beginning of a transactional sequence of actions.
- TCMT delimits the end of a transactional sequence whose actions are to take effect.
- TABT delimits the end of a transactional sequence whose actions are not to take effect.
- ACQ delimits the beginning of a lock issued sequence of actions.
- REL delimits the end of a lock issued sequence of actions.
- NOP is a no operation action. We use this action when a command's reduction results in no work being done, e.g. a thread blocking to wait for a mutex to become acquirable.

The actions R, W, ACQ and REL are parameterised on a memory location ℓ. For example, $\mathsf{R}(\ell)$ denotes that the memory location $\ell \in \mathsf{fs}$ is being read. We use non-parameterised versions of actions when we wish to state that a particular action has been issued but without explicitly stating the concrete semantics of the action. Λ is used to generalise a specific sequence of actions that are executed by each thread within a reduction of a program's non-initialisation commands, i.e. its parallel composition. For example, all reductions in the parallel composition of threads which make progress issue a sequence of actions which conform to the sequence defined by Λ. The use of actions will become clearer as we proceed through this chapter and Chapter 5. At present it is sufficient to understand that every command reduction generates one or more actions from λ, denoted λ^+.

4.2.3.2 Thread

Initialisation Commands Executing the initialisation commands of a thread results in the following $T, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{ld} \xrightarrow{\lambda^+} T', \sigma, \mathsf{fs}', \mathsf{md}, \mathsf{ld}$, where

$$T = \langle \tau, C_{tinit}, C, \mathsf{s}_\tau, \bot \rangle \quad T' = \langle \tau, C'_{tinit}, C, \mathsf{s}_{\tau'}, \bot \rangle$$

Note that only the thread local store and free store components are updated when executing a thread's initialisation command.

Non-Initialisation Commands Executing the non-initialisation commands of a thread results in $T, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{ld} \xrightarrow{\lambda^+} T', \sigma', \mathsf{fs}', \mathsf{md}', \mathsf{ld}'$, where

$$T = \langle \tau, \epsilon, c, \mathsf{s}_\tau, \delta \rangle \quad T' = \langle \tau, \epsilon, c', \mathsf{s}_{\tau'}, \delta' \rangle$$

The reduction results in:

- The thread progressing to the next command in its sequence of non-initialisation commands.
- A possible update of the thread local store and/or global state.
- An update of the redo log δ if c was a transaction.
- An update of md and ld if c was a transaction or lock.
- An update of fs if c performed an allocation.
- The generation of one or more actions drawn from λ.

4.2.3.3 Unified

A reduction of a unified configuration $U \xrightarrow{\lambda^+} U'$, where

$$U = \langle \tau, c, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld} \rangle$$
$$U' = \langle \tau, c', \delta', \mathsf{fs}', \gamma'_{\mathsf{R}}, \gamma'_{\mathsf{W}}, \mathsf{s}'_\tau, \sigma', \mathsf{md}', \mathsf{ld}' \rangle$$

The reduction results in:

- Progression to the next command c'.
- Update of the threaded state δ if c issued a write.
- Update of fs should c have allocated.

- Update of the read and/or respectively write set, γ_{R} and respectively γ_{W}, should c have been executed under a transactional semantics.
- Update of s_τ, σ, md and ld should c be a nested lock.

The following conventions apply when executing a command under a unified configuration:

- c is transactional. The s_τ, σ, md and ld components of a unified configuration are $\perp$.
- c is uncoordinated. The components γ_{R}, γ_{W}, s_τ, σ, md and ld are $\perp$.
- c is a nested lock. All components are defined.

4.2.4 Rules

We now present the rules for the program, thread and unified configurations.

4.2.4.1 Program

Figure 4.4 shows the rules for executing the commands of a program. The rules (PROGRAM−INIT−VAR−DECL) and (PROGRAM−INIT−NEW), which we describe shortly, correspond to label (a) in Figure 4.2.

(PROGRAM−INIT−VAR−DECL) declares a global variable:

- A fresh memory location ℓ is introduced. "fresh" in this context asserts that $\ell \notin \mathsf{fs}$. That is, ℓ is not currently active in the program's free store.

$$\frac{\langle c_1, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^*} \langle c_1', T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma', \mathsf{fs}', \mathsf{md}, \mathsf{Id}\rangle}{\langle c_1; c_2, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^*} \langle c_1'; c_2, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma', \mathsf{fs}', \mathsf{md}, \mathsf{Id}\rangle} \ (\underline{\text{PROGRAM}-\text{INIT}-\text{SEQ}-1})$$

$$\frac{\langle c_1, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^*} \langle \epsilon, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma', \mathsf{fs}', \mathsf{md}, \mathsf{Id}\rangle}{\langle c_1; c_2, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^*} \langle c_2, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma', \mathsf{fs}', \mathsf{md}, \mathsf{Id}\rangle} \ (\underline{\text{PROGRAM}-\text{INIT}-\text{SEQ}-2})$$

$$\frac{\mathsf{fresh}\ \ell \quad \mathsf{s}'{=}\sigma'.\mathsf{s}[v{\mapsto}(\ell, \mathsf{null})] \quad \mathsf{fs}'{=}\mathsf{fs}{\cup}\{\ell\} \quad \sigma'{=}(\mathsf{s}', \sigma.\mathsf{h})}{\langle cn\ v, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\mathsf{NOP}} \langle \epsilon, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma', \mathsf{fs}', \mathsf{md}, \mathsf{Id}\rangle} \ (\underline{\text{PROGRAM}-\text{INIT}-\text{VAR}-\text{DECL}})$$

$$\frac{\begin{array}{c}[v{\mapsto}(\ell, val)]{\subseteq}\sigma.\mathsf{s} \quad (obj, locs){=}\mathsf{CreateObject}(cn, \mathsf{fs}) \quad \mathsf{fs}'{=}\mathsf{fs}{\cup}locs \\ \ell_{\text{base}}{=}\mathsf{Head}(locs) \quad \mathsf{s}'{=}\sigma.\mathsf{s}[v{\mapsto}(\ell, \ell_{\text{base}})] \quad \mathsf{h}'{=}\sigma.\mathsf{h}[\ell_{\text{base}}{\mapsto}obj] \quad \sigma'{=}(\mathsf{s}', \mathsf{h}')\end{array}}{\begin{array}{c}\langle v := \mathsf{new}\ cn, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle \\ \xrightarrow{\mathsf{W}(\ell)} \\ \langle \epsilon, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma', \mathsf{fs}', \mathsf{md}, \mathsf{Id}\rangle\end{array}} \ (\underline{\text{PROGRAM}-\text{INIT}-\text{NEW}})$$

$$\frac{\begin{array}{c}\mathsf{fresh}\ \mathsf{s}_1 \ldots \mathsf{fresh}\ \mathsf{s}_n \\ T_1'{=}\langle 1, C_{1_{tinit}}, C_1, \mathsf{s}_1, \bot\rangle \quad \ldots \quad T_n'{=}\langle n, C_{n_{tinit}}, C_n, \mathsf{s}_n, \bot\rangle\end{array}}{\langle \epsilon, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\mathsf{NOP}} \langle \epsilon, T_1' \mid\mid \ \ldots \ \mid\mid T_n', \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle} \ (\underline{\text{PROGRAM}-\text{FORK}})$$

$$\frac{T_1{=}\langle 1, \epsilon, \epsilon, \mathsf{s}_1, \bot\rangle \quad \ldots \quad T_n{=}\langle n, \epsilon, \epsilon, \mathsf{s}_n, \bot\rangle}{\langle \epsilon, T_1 \mid\mid \ \ldots \ \mid\mid T_n, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\mathsf{NOP}} \langle \epsilon, \epsilon, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\rangle} \ (\underline{\text{PROGRAM}-\text{JOIN}})$$

Figure 4.4: Program Command Rules.

- The updated store s' is the same as $\sigma.\mathsf{s}$ but maps v to the pair (ℓ, null), where the first component of the tuple is v's memory location and the second v's

value.

- ℓ becomes bound in the free store.
- The new global state σ' uses s' as its variable mapping.
- The variable declaration emits a no operation action. Our rules omit a no operation action whenever a reduction has no bearing on read, write or coordination semantics.

We often use the simpler form $\sigma.\mathsf{s}$ and $\sigma.\mathsf{h}$ for addressing the store and heap components of a state, where $\sigma.\mathsf{s} \stackrel{\mathsf{def}}{=} \mathsf{fst}(\sigma)$ and respectively $\sigma.\mathsf{h} \stackrel{\mathsf{def}}{=} \mathsf{snd}(\sigma)$, and $\mathsf{fst}((a,b)) = a$ and $\mathsf{snd}((a,b)) = b$.

(PROGRAM−INIT−NEW) executes an object allocation:

- $\mathsf{CreateObject} \stackrel{\mathsf{def}}{=} \mathsf{Type} \times \mathsf{FS} \rightarrow \mathsf{Object} \times \mathsf{LocationSet}$ returns a tuple whose first component is an object mapping *obj* representing an instance of *cn*, and second component the set of memory locations associated with the fields of *obj*.
- The set of memory locations *locs* consumed by *obj* are bound in the program's free store.
- The head of *locs* is the base location of *obj*. That is, ℓ_{base} is the start address of *obj*. The base address is the memory location associated with *obj*'s first field. Where, $\mathsf{Head}(\{\ell_1, \ldots, \ell_n\})=\ell_1$.
- The updated store mapping s' is the same as $\sigma.\mathsf{s}$ with the exception that the value of v is the base location of *obj*. The updated heap mapping h' is the

same as $\sigma.\mathsf{h}$ but maps ℓ_{base} to the newly created object obj. The updated global state σ' comprises s' and h'.

- Execution of the allocation omits a write action on the memory location of v, $\mathsf{W}(\ell)$.

The object model we use is very simple: each field has an associated distinct memory location; the memory location of an object's first field is its base location. For example, given the class definition `class Coord { Int x; Int y; }`, an object `o` of type `Coord` looks like that shown in Figure 4.5. Essentially, objects have the same memory semantics as structs in C, with the exception that each field has a fixed width of a single memory location.

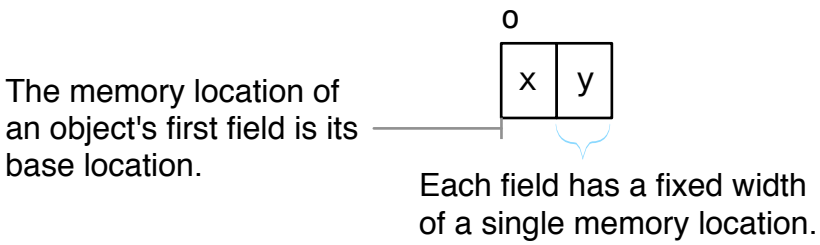


Figure 4.5: The object model used by our semantics.

The rules (PROGRAM−INIT−SEQ−1) and (PROGRAM−INIT−SEQ−2) are applied when executing sequences of initialisation commands drawn from C_{pinit}.

(PROGRAM−FORK) forks the parallel composition of threads upon all the program initialisation commands having been executed. The forking of threads corresponds to label (b) in Figure 4.2:

- A fresh store mapping is created for each thread configuration.

- A thread configuration is initialised for each thread's program text:
  - The thread identifier is a strictly increasing integer.
  - The thread's initialisation commands are the variable declarations from T in Figure 4.1.
  - The thread's non-initialisation commands are the command sequence drawn from the options in C in Figure 4.1.
  - The thread configuration takes on one of the fresh stores.
  - The redo log component δ is initially set to $\perp$.
- The reduction sees a no operation action issued, and the thread configurations of the program being in an active state. That is, the thread configurations begin execution. Thread management activities, e.g. fork and join, do not emit actions.

(PROGRAM−JOIN) performs an n-thread join when all threads have finished executing their respective commands. This rule corresponds to label (e) in Figure 4.2:

- The thread configurations $T_1 \ldots T_n$ have executed all of their respective initialisation and non-initialisation commands. This is indicated by the initialisation commands being ϵ and the non-initialisation commands being ϵ in each respective thread configuration.
- The reduced program configuration uses ϵ for the value of the parallel composition component of the program configuration. Here, ϵ indicates that all threads have completed their execution. The program implicitly terminates.

We have explained most of the program execution lifetime. However, we have not described the rule that governs reductions while each thread is executing its non-initialisation commands in parallel with respect to the non-initialisation commands being executed by the other active threads of the parallel composition (label (d) in Figure 4.2). We defer coverage of this topic until Section 4.2.5 as it requires an understanding of the rules given in Sections 4.2.4.2 and 4.2.4.3.

4.2.4.2 Thread

We now present the rules that execute the commands that correspond to label (d) in Figure 4.2. The thread rules are given in Figures 4.6, 4.7, 4.8 and 4.9. A thread executes a sequence of initialisation commands (thread-local variable declarations), then a sequence of non-initialisation commands (any command in C in Figure 4.1). At any given point of a thread's execution a non-initialisation command is being executed under one of three coordination semantics: uncoordinated, lock or transactional. The actual execution of each command is performed by the unified rules given in Section 4.2.4.3. The purpose of the rules that execute the non-initialisation commands of a thread is to setup the execution context for for a command to be executed under the unified rules. Recall that the unified rules permit a single definition of all commands, irrespective of their executing coordination semantics. This single definition for each command comes at the cost of slightly reducing the intuitiveness of the thread rules.

(THREAD−INIT−VAR−DECL) executes a variable declaration as part of a thread's initialisation commands. The semantics are the same as the program rule (PROGRAM−INIT−VAR−DECL) with the exception that the variable is added to the domain of s_τ, the thread local store mapping. (THREAD−INIT−SEQ−ONE)

$$\frac{\begin{array}{c}(\underline{\mathsf{THREAD{-}INIT{-}VAR{-}DECL}})\\ \mathsf{fresh}\ \ell \quad \mathsf{fs}'{=}\mathsf{fs}{\cup}\{\ell\} \quad \mathsf{s}'_\tau{=}\mathsf{s}_\tau[v{\mapsto}(\ell,\mathsf{null})]\end{array}}{\langle \tau, cn\ v, C, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\mathsf{NOP}} \langle \tau, \epsilon, C, \mathsf{s}'_\tau, \bot\rangle, \sigma, \mathsf{fs}', \mathsf{md}, \mathsf{Id}}$$

$$\frac{\langle \tau, c_1, C, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\lambda^+} \langle \tau, c'_1, C, \mathsf{s}'_\tau, \bot\rangle, \sigma, \mathsf{fs}', \mathsf{md}, \mathsf{Id}}{\langle \tau, c_1; c_2, C, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\lambda^+} \langle \tau, c'_1; c_2, C, \mathsf{s}'_\tau, \bot\rangle, \sigma, \mathsf{fs}', \mathsf{md}, \mathsf{Id}}\ (\underline{\mathsf{THREAD{-}INIT{-}SEQ{-}ONE}})$$

$$\frac{\langle \tau, c_1, C, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\lambda^+} \langle \tau, \epsilon, C, \mathsf{s}'_\tau, \bot\rangle, \sigma, \mathsf{fs}', \mathsf{md}, \mathsf{Id}}{\langle \tau, c_1; c_2, C, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\lambda^+} \langle \tau, c_2, C, \mathsf{s}'_\tau, \bot\rangle, \sigma, \mathsf{fs}', \mathsf{md}, \mathsf{Id}}\ (\underline{\mathsf{THREAD{-}INIT{-}SEQ{-}TWO}})$$

$$\frac{\langle \tau, \epsilon, c_1, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\lambda^+} \langle \tau, \epsilon, c'_1, \mathsf{s}'_\tau, \bot\rangle, \sigma', \mathsf{fs}', \mathsf{md}', \mathsf{Id}'}{\langle \tau, \epsilon, c_1; c_2, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\lambda^+} \langle \tau, \epsilon, c'_1; c_2, \mathsf{s}'_\tau, \bot\rangle, \sigma', \mathsf{fs}', \mathsf{md}', \mathsf{Id}'}\ (\underline{\mathsf{THREAD{-}SEQ{-}ONE}})$$

$$\frac{\langle \tau, \epsilon, c_1, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\lambda^+} \langle \tau, \epsilon, \epsilon, \mathsf{s}'_\tau, \bot\rangle, \sigma', \mathsf{fs}', \mathsf{md}', \mathsf{Id}'}{\langle \tau, \epsilon, c_1; c_2, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\lambda^+} \langle \tau, \epsilon, c_2, \mathsf{s}'_\tau, \bot\rangle, \sigma', \mathsf{fs}', \mathsf{md}', \mathsf{Id}'}\ (\underline{\mathsf{THREAD{-}SEQ{-}TWO}})$$

$$\frac{\begin{array}{c}c \neq \mathsf{sync}(\_)\{\_\} \wedge c \neq \mathsf{atomic}\{\_\}\\ \delta{=}(\mathsf{s}_\tau \cup \sigma.\mathsf{s}, \sigma.\mathsf{h})\\ \langle \tau, c, \delta, \mathsf{fs}, \bot, \bot, \bot, \bot, \bot, \bot\rangle \xrightarrow{\lambda^+} \langle \tau, c', \delta', \mathsf{fs}', \bot, \bot, \bot, \bot, \bot, \bot\rangle\\ (\mathsf{s}'_\tau, \sigma'){=}\mathsf{Persist}(\delta', \mathsf{s}_\tau, \sigma)\end{array}}{\langle \tau, \epsilon, c, \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id} \xrightarrow{\lambda^+} \langle \tau, \epsilon, c', \mathsf{s}'_\tau, \bot\rangle, \sigma', \mathsf{fs}', \mathsf{md}, \mathsf{Id}}\ (\underline{\mathsf{THREAD{-}UNCOORDINATED}})$$

Figure 4.6: Thread Command Rules (Part I).

and (THREAD−INIT−SEQ−TWO) execute each command within a thread's sequence of initialisation commands. When all of a thread's initialisation com-

$$\frac{\begin{array}{c}(\underline{\text{THREAD}-\text{TRANSACTION}-\text{BEGIN}}) \\ id'{=}\mathsf{GenerateID}(\mathsf{md},\mathsf{Id}) \\ \mathsf{md}'{=}\mathsf{md}[id'{\mapsto}(\mathsf{Now}(),\bot,\{\},\{\},\{\},\mathcal{A})] \\ \delta{=}(\mathsf{s}_\tau \cup \sigma.\mathsf{s},\sigma.\mathsf{h})\end{array}}{\begin{array}{c}\langle \tau,\epsilon,\mathsf{id}{:}\mathsf{atomic}\{c\},\mathsf{s}_\tau,\bot\rangle,\sigma,\mathsf{fs},\mathsf{md},\mathsf{Id} \\ \xrightarrow{\text{TBEG}} \\ \langle \tau,\epsilon,id'{:}\mathsf{ablk}(c,\mathsf{id}{:}\mathsf{atomic}\{c\}),\mathsf{s}_\tau,\delta\rangle,\sigma,\mathsf{fs},\mathsf{md}',id'\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\text{THREAD}-\text{TRANSACTION}-\text{IN}}) \\ [id{\mapsto}(\mathsf{beg},\bot,\gamma_\mathsf{R},\gamma_\mathsf{W},\gamma_\mathsf{D},\mathsf{coord})]{\subseteq}\mathsf{md} \\ \langle \tau,c,\delta,\mathsf{fs},\gamma_\mathsf{R},\gamma_\mathsf{W},\bot,\bot,\bot,\bot\rangle \xrightarrow{\lambda^+_{RW}} \langle \tau,c',\delta',\mathsf{fs}',\gamma'_\mathsf{R},\gamma'_\mathsf{W},\bot,\bot,\bot,\bot\rangle \\ \mathsf{md}'{=}\mathsf{md}[id{\mapsto}(\mathsf{beg},\mathsf{cmt},\gamma'_\mathsf{R},\gamma'_\mathsf{W},\gamma'_\mathsf{R}{\cup}\gamma'_\mathsf{W},\mathsf{coord})]\end{array}}{\begin{array}{c}\langle \tau,\epsilon,id{:}\mathsf{ablk}(c,\overleftarrow{c}),\mathsf{s}_\tau,\delta\rangle,\sigma,\mathsf{fs},\mathsf{md},\mathsf{Id} \\ \xrightarrow{\lambda^+_{RW}} \\ \langle \tau,\epsilon,id{:}\mathsf{ablk}(c',\overleftarrow{c}),\mathsf{s}_\tau,\delta'\rangle,\sigma,\mathsf{fs}',\mathsf{md}',\mathsf{Id}\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\text{THREAD}-\text{TRANSACTION}-\text{COMMIT}}) \\ \neg\mathsf{Conflict}(id,\mathsf{md}) \\ [id{\mapsto}(\mathsf{beg},\bot,\gamma_\mathsf{R},\gamma_\mathsf{W},\gamma_\mathsf{D},\mathsf{coord})]{\subseteq}\mathsf{md} \\ \mathsf{md}'{=}\mathsf{md}[id{\mapsto}(\mathsf{beg},\mathsf{Now}(),\gamma_\mathsf{R},\gamma_\mathsf{W},\gamma_\mathsf{D},\mathsf{coord})] \\ (\mathsf{s}'_\tau,\sigma'){=}\mathsf{Persist}(\delta,\mathsf{s}_\tau,\sigma)\end{array}}{\begin{array}{c}\langle \tau,\epsilon,id{:}\mathsf{ablk}(\epsilon,\overleftarrow{c}),\mathsf{s}_\tau,\delta\rangle,\sigma,\mathsf{fs},\mathsf{md},\mathsf{Id} \\ \xrightarrow{\text{TCMT}} \\ \langle \tau,\epsilon,\epsilon,\mathsf{s}'_\tau,\bot\rangle,\sigma',\mathsf{fs},\mathsf{md}',\mathsf{Id}\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\text{THREAD}-\text{TRANSACTION}-\text{ABORT}}) \\ \mathsf{Conflict}(id,\mathsf{md}) \\ \mathsf{md}'{=}\mathsf{md} \quad \mathsf{Dom}(\mathsf{md}'){=}\mathsf{Dom}(\mathsf{md}')\backslash\{id\}\end{array}}{\begin{array}{c}\langle \tau,\epsilon,id{:}\mathsf{ablk}(c,\overleftarrow{c}),\mathsf{s}_\tau,\delta\rangle,\sigma,\mathsf{fs},\mathsf{md},\mathsf{Id} \\ \xrightarrow{\text{TABT}} \\ \langle \tau,\epsilon,\overleftarrow{c},\mathsf{s}_\tau,\bot\rangle,\sigma,\mathsf{fs},\mathsf{md}',\mathsf{Id}\end{array}}$$

Figure 4.7: Thread Command Rules (Part II).

$$\frac{\begin{array}{c}(\underline{\textsf{THREAD}-\textsf{LOCK}-\textsf{ACQUIRE}})\\ \ell{=}\textsf{VarLocation}(\textsf{s}_\tau, \sigma, v)\\ \textsf{Acquireable}(\ell, \textsf{md})\\ id'{=}\textsf{GenerateID}(\textsf{md}, \textsf{Id})\\ \textsf{md}'{=}\textsf{md}[id' \mapsto (\textsf{Now}(), \bot, \{\}, \{\}, \{\ell\}, \mathcal{L}(\tau, 1))]\end{array}}{\begin{array}{c}\langle \tau, \epsilon, \textsf{id}{:}\textsf{sync}(v)\{c\}, \textsf{s}_\tau, \bot\rangle, \sigma, \textsf{fs}, \textsf{md}, \textsf{Id}\\ \xrightarrow{\textsf{ACQ}(\ell)}\\ \langle \tau, \epsilon, id'{:}\textsf{sblk}(c), \textsf{s}_\tau, \bot\rangle, \sigma, \textsf{fs}, \textsf{md}', id'\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\textsf{THREAD}-\textsf{LOCK}-\textsf{RELEASE}})\\ [id \mapsto (\textsf{beg}, \bot, \gamma_\textsf{R}, \gamma_\textsf{W}, \{\ell\}, \mathcal{L}(\tau, 1))] \subseteq \textsf{md}\\ \textsf{md}'{=}\textsf{md}[id \mapsto (\textsf{beg}, \textsf{Now}(), \gamma_\textsf{R}, \gamma_\textsf{W}, \{\}, \mathcal{L}(\tau, 0))]\end{array}}{\begin{array}{c}\langle \tau, \epsilon, id{:}\textsf{sblk}(\epsilon), \textsf{s}_\tau, \bot\rangle, \sigma, \textsf{fs}, \textsf{md}, \textsf{Id}\\ \xrightarrow{\textsf{REL}(\ell)}\\ \langle \tau, \epsilon, \epsilon, \textsf{s}_\tau, \bot\rangle, \sigma, \textsf{fs}, \textsf{md}', \textsf{Id}\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\textsf{THREAD}-\textsf{LOCK}-\textsf{BLOCKING}})\\ \ell{=}\textsf{VarLocation}(\textsf{s}_\tau, \sigma, v)\\ \neg\textsf{Acquireable}(\ell, \textsf{md})\end{array}}{\begin{array}{c}\langle \tau, \epsilon, \textsf{id}{:}\textsf{sync}(v)\{c\}, \textsf{s}_\tau, \bot\rangle, \sigma, \textsf{fs}, \textsf{md}, \textsf{Id}\\ \xrightarrow{\textsf{NOP}}\\ \langle \tau, \epsilon, \textsf{id}{:}\textsf{sync}(v)\{c\}, \textsf{s}_\tau, \bot\rangle, \sigma, \textsf{fs}, \textsf{md}, \textsf{Id}\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\textsf{THREAD}-\textsf{LOCK}-\textsf{IN}})\\ c \neq \textsf{sync}(\_)\{\_\}\\ \delta{=}(\textsf{s}_\tau \cup \sigma.\textsf{s}, \sigma.\textsf{h})\\ \langle \tau, c, \delta, \textsf{fs}, \bot, \bot, \bot, \bot, \bot, \bot\rangle \xrightarrow{\lambda^+} \langle \tau, c', \delta', \textsf{fs}', \bot, \bot, \bot, \bot, \bot, \bot\rangle\\ (\textsf{s}'_\tau, \sigma'){=}\textsf{Persist}(\delta', \textsf{s}_\tau, \sigma)\end{array}}{\begin{array}{c}\langle \tau, \epsilon, id{:}\textsf{sblk}(c), \textsf{s}_\tau, \bot\rangle, \sigma, \textsf{fs}, \textsf{md}, \textsf{Id}\\ \xrightarrow{\lambda^+}\\ \langle \tau, \epsilon, id{:}\textsf{sblk}(c'), \textsf{s}'_\tau, \bot\rangle, \sigma', \textsf{fs}', \textsf{md}, \textsf{Id}\end{array}}$$

Figure 4.8: Thread Command Rules (Part III).

$$\frac{\begin{array}{c}(\underline{\mathsf{THREAD{-}LOCK{-}IN{-}LOCK}})\\ c = \mathsf{sync}(\_)\{\_\}\\ \delta{=}(\mathsf{s}_\tau \cup \sigma.\mathsf{s}, \sigma.\mathsf{h})\\ \langle \tau, c, \delta, \mathsf{fs}, \bot, \bot, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, c', \delta', \mathsf{fs}', \bot, \bot, \mathsf{s}'_\tau, \sigma', \mathsf{md}', \mathsf{Id}'\rangle\end{array}}{\begin{array}{c}\langle \tau, \epsilon, id{:}\mathsf{sblk}(c), \mathsf{s}_\tau, \bot\rangle, \sigma, \mathsf{fs}, \mathsf{md}, \mathsf{Id}\\ \xrightarrow{\lambda^+}\\ \langle \tau, \epsilon, id{:}\mathsf{sblk}(c'), \mathsf{s}'_\tau, \bot\rangle, \sigma', \mathsf{fs}', \mathsf{md}', \mathsf{Id}'\end{array}}$$

Figure 4.9: Thread Command Rules (Part IV).

mands have been executed, i.e. reduced to the empty command ϵ, the thread's non-initialisation commands are executed, which we cover from this point forward.

($\underline{\mathsf{THREAD{-}SEQ{-}ONE}}$) and ($\underline{\mathsf{THREAD{-}SEQ{-}TWO}}$) execute the commands within a thread's non-initialisation sequence of commands. Note that each command's execution can result in the update of s_τ, σ, fs, md and Id. By contrast, a command executed as part of the thread's initialisation commands only updates s_τ and fs.

($\underline{\mathsf{THREAD{-}UNCOORDINATED}}$) executes a command under an uncoordinated semantics:

- The command c is not a lock or a transaction.
- The unified configuration that c is executed under contains the thread identifier of the thread executing c, a state whose store component is the union of the thread store and global store, and second component the global heap. Where, $\mathsf{s}_\tau \cup \sigma.\mathsf{s}$ unifies the domain and co-domain of the store mappings s_τ and $\sigma.\mathsf{s}$. $\mathsf{Persist} \stackrel{\mathsf{def}}{=} \mathsf{State} \times \mathsf{Store} \times \mathsf{State} \to \mathsf{Store} \times \mathsf{State}$ persists the effect

of a command's mutations. The first argument of Persist is the state we wish to persist, and the remaining arguments the store and state we wish to persist the effect into (the thread-local store and the global state). The returned tuple is a store and global state with the effect of c's execution persisted.

- Executing the command sees a number of actions being issued. The exact actions will be defined when we cover the unified rules.

(THREAD−TRANSACTION−BEGIN) begins the execution of a transaction:

- A unique integer id' is created via $\mathsf{GenerateID}(\mathsf{md}, \mathsf{Id})$ which generates the next unique integer identifier. id' is no longer a candidate for future unique labels so Id is replaced with id' in the reduction. Note that the definition of $\mathsf{GenerateID} \stackrel{\mathsf{def}}{=} \mathsf{MD} \times \mathsf{ID} \to \mathsf{ID}$ is trivial – it simply returns the successor of Id and checks that the successor is not in the domain of md.

- The language construct $\mathsf{id{:}atomic}\{c\}$ is translated to the intermediate construct $id'{:}\mathsf{ablk}(c, \mathsf{id{:}atomic}\{c\})$. The first component of ablk is the command the transaction is to execute and the second component the point at which the program counter should rollback to should the transaction abort. We refer to the point of rollback in subsequent rules as $\overleftarrow{c}$.

- The medadata mapping md is updated to reflect the newly initiated transaction's state: the time at which it began, Now(), and the fact that the coordination instance the metadata models is that of a transaction, $\mathcal{A}$. All other components of the metadata entry are initialised to their default components: {} for the read, write and dataset and $\perp$ for the transaction's

commit time. The function Now yields an integer timestamp marking the current point in time.

- The transaction's effect, its *redo log*, is stored in δ which is a state pair whose first component is that of the thread local store and global store combined, and second component that of the global heap.

(THREAD−TRANSACTION−IN) executes a command c transactionally:

- We assert the current metadata that id, the unique identifier associated with the current transactional instance, maps to in md. Note that in the assertion $[id{\mapsto}(\mathsf{beg}, \mathsf{cmt}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \gamma_{\mathsf{D}}, \mathsf{coord})]{\subseteq}\mathsf{md}$ we use the canonical labels beg, cmt, γ_{R}, γ_{W}, γ_{D} and coord to bind to the respective component's current value in id's metadata. When we do not wish to bind the current value, e.g. we want to set or assert existence of a specific value within id's metadata, we use a permissible value of that component's type. For example, in $[id{\mapsto}(\mathsf{beg}, \bot, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \gamma_{\mathsf{D}}, \mathsf{coord})]{\subseteq}\mathsf{md}$ we assert that the coordination instance with identifier id has yet to complete, and in $\mathsf{md}'{=}\mathsf{md}[id{\mapsto}(\mathsf{beg}, \mathsf{Now}(), \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \gamma_{\mathsf{D}}, \mathsf{coord})]$ we are setting the value of id's commit time.

- The command c is executed via our unified command configuration which we cover in Section 4.2.4.3. The main things of note are that we use the transaction's redo log as the state under which c is executed in addition to incrementally building the read and write set of the transaction. The remaining components of the unified configuration are irrelevant for executing c transactionally.

- The new value of the metadata which id maps to in md' differs in its read, write and dataset to the value it mapped to in md. A transaction's dataset is incrementally built on a per-command basis. The dataset of a transaction is validated pre-commit, rather than incrementally.

(THREAD−TRANSACTION−COMMIT) commits a transaction:

- The transaction can be committed if the predicate Conflict fails. That is, if the write set of id does not conflict with the dataset of any recently ran or still running transaction or lock.
- The metadata entry for id is updated to reflect its commit time.
- The effect of δ is persisted by merging its effect into the appropriate component of σ and s_τ via Persist.
- The redo log of the committed transaction is discarded.

(THREAD−TRANSACTION−ABORT) aborts a transaction:

- The transaction may not be committed as it conflicts with another running or recently ran lock or transaction.
- The metadata associated with id is removed from md.
- The transaction's redo log is discarded.
- The program counter of the thread executing the aborted transaction is set to its rollback command $\overleftarrow{c}$. Recall that the rollback command refers to the transaction. That is, the transaction is simply retried until it is eventually permitted to commit.

The thread rules that execute a lock represent the execution of the most parent lock. (THREAD−LOCK−ACQUIRE) initiates the execution of a lock if its mutex can be acquired:

- $\mathsf{VarLocation} \stackrel{\mathsf{def}}{=} \mathsf{Store} \times \mathsf{State} \times \mathsf{Variable} \to \mathsf{Location}$ gets the memory location ℓ of the variable v being used as the mutex.
- $\mathsf{Acquireable} \stackrel{\mathsf{def}}{=} \mathsf{Location} \times \mathsf{MD} \to \mathsf{Bool}$ is a predicate that is true only if an actively executing lock has not already acquired ℓ.
- A unique identifier for the lock instance is generated via GenerateID. The generated identifier is no longer unique, so becomes the current value of Id in the reduction.
- The metadata mapping is updated to contain a new entry for the now active lock instance. Its begin time is set via Now, its dataset is initialised to the location of the mutex it is protected on and the coordination type the metadata represents is labelled as $\mathcal{L}(\tau, 1)$ to denote that the metadata models a lock whose mutex is owned by thread τ and the handle count on that mutex is 1. A lock never has a read or write set, only a dataset. The dataset of a lock comprises the mutex the lock instance has acquired. Because (THREAD−LOCK−ACQUIRE) always executes the most parent lock the handle count will always be set to 1 upon an acquisition.
- An acquire action is issued parameterised on the location of the mutex.
- The reduction features the use of the intermediate construct sblk which takes on the unique identifier id'.

(THREAD−LOCK−RELEASE) applies when all of a lock's constituent commands have been executed. Again, because we are executing the most parent lock the handle count will always be 1 when releasing a mutex.

- The premise begins by asserting that id, the unique label associated with the current lock instance, is yet to complete and the handle count on the mutex ℓ is 1. A lock is only ever released when the handle count on the mutex is 1.
- ℓ is the location of the mutex used by the lock. Reduction of the thread configuration results in the generation of a release instruction on ℓ.
- The metadata mapping is updated to reflect the time of lock instance id's completion time, the removal of ℓ from id's dataset and the handle count being set to 0. The last two components have no logical impact on our overall system but they provide a visual cue to the releasing of a resource.
- The location of the mutex v, ℓ, cannot be acquired due to Acquireable failing. That is, ℓ is acquired by a currently running lock in a thread other than τ.
- The thread reduction sees no change in the thread's program text. Here, the effect is that the thread appears to continually try to acquire ℓ until at some stage ℓ becomes available and (THREAD−LOCK−ACQUIRE) may be applied.
- The reduction sees the generation of the action NOP.

(THREAD−LOCK−IN) executes a command under a lock semantics as long as the command is not a lock.

- c is not a lock. Note that $\mathsf{sync}(\_)\{\_\}$ denotes we have no interest in the mutex or command the lock is defined on, only that c is a lock.

- c is executed with no read or write set being accumulated. Recall that a lock has no need for a read or write set. The remaining components of the unified configuration are not required as the current lock is in charge of persisting the effect of c to memory.

- The effect of c is persisted immediately to the thread-local store and global state. This is in contrast to transactions where multiple writes and reads may have occurred before such actions are observable by other threads.

($\underline{\mathsf{THREAD{-}LOCK{-}IN{-}LOCK}}$) executes a nested lock:

- The command c (the nested lock) is executed under a unified configuration that is given s_τ, σ, md and ld. These components are specified because it is the task of the nested lock to persist its effect to these components, not the parent lock. (We revisit this point shortly.)

- The reduction sees the thread configuration taking on the updated values of s_τ, σ, md and ld. s'_τ and σ' comprise the effect of the nested lock's commands, md' the nested lock's supporting metadata and ld' the next free unique label.

The details of nested locks will be clearer upon reading Section 4.2.4.3. However, we now give a conceptual overview of effect persistence with respect to parent and child locks. The key point is that a lock executing a non-lock command is in charge of persisting the effect of its immediate command; however,

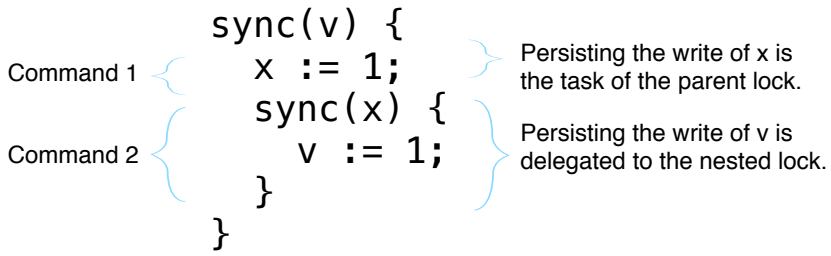


Figure 4.10: The parent lock contains two commands: a write of `x` and a lock. The nested lock contains a write of `v`. The most nested active lock is in charge of persisting the effect of its commands. For example, the parent lock persists the write of `x`, while the nested lock is in charge of persisting the write of `v`.

if a lock is executing a lock, then persisting the effect of the nested lock's commands is delegated to the nested lock. The latter point is shown in the reduction of (THREAD−LOCK−IN−LOCK) where the reduction takes the values of s'_τ, σ', md' and ld' from the reduced unified configuration. By contrast, the updated values for s_tid and σ in the reduction of (THREAD−LOCK−IN) are constructed directly. Figure 4.10 gives a general intuition as to which lock is in charge of persisting the effect of a command. Here, the parent lock executes its first command using (THREAD−LOCK−IN) and second command, the nested lock, with (THREAD−LOCK−IN−LOCK). The persisted effect of a nested lock bubbles up until it reaches the most parent lock, one initiated via (THREAD−LOCK−ACQUIRE). The recursive nature of locks is extended further in Section 4.2.4.3.

4.2.4.3 Unified Commands

The unified commands are given in Figures 4.11, 4.12, 4.13, 4.14 and 4.15. All commands are defined in terms of a unified configuration.

(UNIFIED−NESTED−LOCK−ACQUIRE) is applied when a lock is executing a nested lock and is similar to (THREAD−LOCK−ACQUIRE). A nested lock is at first a consequence of executing the rule (THREAD−LOCK−IN−LOCK) but is also applied as a consequence of (UNIFIED−NESTED−LOCK−IN−LOCK).

(UNIFIED−NESTED−LOCK−ACQUIRE−REC) is applied when a nested lock wishes to acquire a mutex which is already held by a parent lock executed by the same thread:

- The memory location of the mutex v is asserted to be not acquirable as it is held by an active lock, in addition it is asserted that the mutex is held by the current thread τ.
- The existential states that there exists an actively executing lock in md such that it uses the same mutex that the nested lock wishes to acquire, is running on the same thread and has a handle count greater than or equal to one.
- The handle count of the mutex is incremented.
- The nested locks recycles the identifier of the original lock which acquired the mutex. Intuitively, the nested lock does not alter the semantics of the original acquiring lock, so there is no need to treat the recursive lock instance as being logically distinct.
- An acquire action is issued during the reduction.

(UNIFIED−NESTED−LOCK−BLOCKING) is almost identical to (THREAD−LOCK−BLOCKING) and is applied when a nested lock cannot acquire its mutex.

(UNIFIED−NESTED−LOCK−RELEASE−REC) is applied when a lock releases a recursively acquired mutex:

- The handle count on the mutex to be released by the lock is greater than one. This implies that the lock was recursively acquired and that there exists a parent lock that still requires the mutex be held by the thread.
- The handle count on the mutex is decremented.
- The reduction sees a release action being generated on the mutex.

(UNIFIED−NESTED−LOCK−RELEASE) is applied when a mutex acquired by a nested lock can be released:

- The handle count associated with the mutex to be released is one. That is, the current nested lock instance is the last instance that has a use for the mutex.
- The updated metadata instance sees the identifier of the releasing lock removing the mutex from its dataset component and setting its handle count to zero.
- The reduction issues a release action on the mutex.

(UNIFIED−NESTED−LOCK−IN) is similar to (THREAD−LOCK−IN). Here, the command being executed by a nested lock is not a lock, so the responsibility of persisting the effect of c is the task of the immediate lock instance. (UNIFIED−NESTED−LOCK−IN−LOCK) is similar to (THREAD−LOCK−IN−LOCK) in that the responsibility of persisting the child lock's commands are that of the

child lock, not the immediate lock. The persisted effect of each of the nested lock's commands bubble up to the unified configuration executing the parent lock.

The rules governing nested locks are tricky to understand so we now provide a general summary of their operation in Figure 4.16. Let us assume that lock instance 1 is a nested lock. A general overview of the relevant rule applications follows. The first command of nested lock instance 1 is a lock. Consequently, the rule (UNIFIED−NESTED−LOCK−IN−LOCK) is applied. Assuming the nested lock can acquire its non-recursive mutex it applies (UNIFIED−NESTED−LOCK−ACQUIRE), followed by (UNIFIED−NESTED−LOCK−IN) as the lock's first command is not a lock. Lock instance 2 then applies (UNIFIED−NESTED−LOCK−RELEASE) and passes the effect of its assignment back to lock instance 1 in the form of an updated thread store and global state. Note also that the nested lock passes back an updated metadata mapping and identifier component as the nested lock mutated them during its execution. Lock instance 1 then executes its second command which is an assignment via (UNIFIED−NESTED−LOCK−IN) followed by an application of (UNIFIED−NESTED−LOCK−RELEASE). Lock instance 1 then passes the effect of executing its commands to its parent lock, and so on until control returns to the most parent lock instance.

(UNIFIED−ASSIGN) assigns the value of one variable to another.

- The updated store s' sees v take on x's value. Note that we often use placeholders such as val_v and val_x when we do not care what the value of a particular variable or field is.

- The assertion $\ell_1 \neq \ell_2$ denotes that `v` and `x` occupy different stack slot locations.

- The update state δ' comprises the updated store s' but the heap component remains the same as $\delta.\mathsf{h}$.
- The updated read set comprises x's memory location ℓ_2; the updated write set comprises v's memory location ℓ_1.
- The reduction results in a read instruction on ℓ_2 and write instruction on ℓ_1.

(UNIFIED−FLD−UPD) updates the value of a field to be that of a variable.

- The value of v must be a memory location that is not equal to that of the physical locations of v and x. This assertion maintains the invariant that the stack and heap memory pools are logically distinct.
- The value of v must be in the domain of $\delta.\mathsf{h}$.
- The location of the field f is attained via $\mathsf{FldLoc} \stackrel{\mathsf{def}}{=} \mathsf{State} \times \mathsf{Variable} \times \mathsf{Field} \rightarrow \mathsf{Location}$.
- We update the value of f to that of x's value via $\mathsf{FldUpd} \stackrel{\mathsf{def}}{=} \mathsf{State} \times \mathsf{Variable} \times \mathsf{Field} \times \mathsf{Location} \rightarrow \mathsf{Heap}$. The returned heap mapping entails f in the object that $v.f$ refers to having the value val_x.
- The update state δ' comprises the old store and the updated heap.
- The updated read set comprises the locations of v and x; the updated write set comprises the location of f.
- The reduction sees read actions issued on v and x and a write action on f.

(UNIFIED−ASSIGN−FLD) assigns the value of a field to a variable:

- The value of f is attained via $\mathsf{FldVal} \stackrel{\mathsf{def}}{=} \mathsf{State} \times \mathsf{Variable} \times \mathsf{Field} \to \mathsf{Location}$.
- The updated store s' sees v taking on the value of f. The updated state comprises s' and $\delta.\mathsf{h}$ as executing the command only updates the store.
- The updated read set comprises a read on x and f; the updated write set comprises a write on v.
- The reduction sees read actions issued for x and f and a write action for v.

(UNIFIED−NEW) allocates a new object and is generally identical to the program rule (PROGRAM−INIT−NEW) with the exception that (UNIFIED−NEW) updates the write set.

(UNIFIED−EQ) checks whether the value of v is null.

- The predicate $\mathsf{IsNull} \stackrel{\mathsf{def}}{=} \mathsf{Location} \to \mathsf{Bool}$ determines if the value of v is null.
- The reduction goes to the results of the IsNull test.
- A read action on v in the reduction is generated.

(UNIFIED−NEQ) is the same as (UNIFIED−EQ) but checks for inequality with null. (UNIFIED−IF) evaluates the boolean command b. Evaluating a boolean command only every results in the issue of a read, so only the read set is updated. (UNIFIED−IF−TRUE) and (UNIFIED−IF−FALSE) are applied when the boolean b reduces to the canonical values True and respectively False. (UNIFIED−WHILE), (UNIFIED−WHILE−TRUE) and (UNIFIED−WHILE−FALSE) are similar to the if rules.

(UNIFIED−METHOD−CALL) executes a method:

- The program text of the method is retrieved via $\mathsf{MethodCmds}$ which takes the receiver type and the method name and returns the methods program text. The set of formal arguments a method takes is attained via $\mathsf{FormalArgs}$. We assume this information is easily derivable from the program text. Note that p^* represents zero-or-more arguments. We interpret this as a set: if no arguments are given then p^* is the empty set when calling $\mathsf{PassByValue}$, otherwise it comprises a set of variables which were passed to m.

- $\mathsf{PassByValue} \stackrel{\mathsf{def}}{=} \mathsf{State} \times \mathsf{FS} \times \mathsf{Variable} \times \mathsf{VariableSet} \times \mathsf{VariableSet} \rightarrow \mathsf{Store} \times \mathsf{LocationSet}$ returns a tuple whose first component comprises a store populated with method local variables whose names and values are the same as those passed to the method, and second component the memory locations that the method local variable occupy. Note that the returned store also comprises the special variable *this* whose value is a reference to the base location of the object the method invoked upon.

- The updated read set comprises the memory locations of the variables passed to the method. $\mathsf{ArgLocs} \stackrel{\mathsf{def}}{=} \mathsf{State} \times \mathsf{VariableSet} \rightarrow \mathsf{LocationSet}$ returns the memory locations for the passed in variables.

- The intermediate construct $\mathsf{frame}(c, \mathsf{s})$ is used to delimit the method's program text. The second component of frame is the store to "pop" back in, which is the store of the invoker of the method.

- The new state δ' which to execute the method's program text under comprises the method local store s_m.

- Invoking the method sees a read instruction issued on each of the memory locations of the variables passed in as arguments to the method.

(UNIFIED−METHOD−IN) executes a command of a method. Returning from a method via (UNIFIED−METHOD−RETURN) is trivial – it simply restores the caller's store.

4.2.5 Parallel Composition

Figure 4.17 shows the rule which governs the progress each thread makes during the parallel execution of each thread's non-initialisation commands (label (d) in Figure 4.2).

4.2.5.1 Intuition

At any given time a thread is executing a command under one of three coordination semantics: uncoordinated, lock or transactional. Each thread within the parallel composition makes some form of progress in their respective transition system: a thread executing an uncoordinated command always makes positive progress; a thread executing a transaction makes positive progress if its transaction commits; and a thread executing a lock makes positive progress if it has acquired its respective mutex. *Positive progress* denotes reduction to a thread configuration whose active command succeeds that which was originally executed at the beginning of the program reduction. Threads that make positive progress contribute to the new program state upon a program reduction. Threads that execute a transaction that has been aborted or are blocking waiting for a mutex to become acquirable make *negative progress*. That is, they appear to make no

progress in their respective transition systems.

4.2.5.2 Discussion

(PROGRAM−PARALLEL−COMPOSITON) is a big-step semantics for performing a program reduction while executing the commands of the threads within a parallel composition. Most of the details discussed shortly have already been presented in Section 4.2.4.2. Boxes and labels are used to group like components within the premise of (PROGRAM−PARALLEL−COMPOSITON) to facilitate their discussion.

Label A states that the active set of threads are partitioned into the groups of threads I, J, K, M and U. The threads are partitioned based upon the coordination semantics they are executing: threads in I are executing locks which are blocking; J are those which have acquired their respective mutex; those in K are executing transactions to be aborted; M those executing transactions to be committed; and those in U are executing their respective command under no coordination semantics. This partitioning covers all the semantics of our thread rules given in Section 4.2.4.2. We assume that each thread in J acquires a distinct mutex and that all transactions executed by the threads in M do not conflict with one another. The label comparisons for id_j, id_k and id_m assert that they are valid unique values within the range Id and Id'. We use these labels later when we specify the thread configurations each thread transitions through.

The box labelled B comprises the thread configurations that each thread in I, J, K, M and U transitions through. Box C uses the thread configurations constructed in B to form the relevant program reductions:

- All threads in I have one configuration as a lock that blocks reduces to

the same thread configuration. The reduction does not affect any program component.

- Threads in U make positive progress and reduce to a thread configuration whose command to execute is the one that succeeds the just executed command. The side effect of executing an uncoordinated command can be the update of a thread store, program state and/or free store. No thread in U will update the metadata or coordination instance identifier components.

- Threads in J, the threads executing locks which have managed to acquire their respective mutex, transition through the following configurations:

  - The first reduction to T'_j sees the thread acquire its respective mutex. This action results in an update of md_j and ld_j.

  - A number of intermediate transitions take place as the lock executes its constituent commands. We denote this via $(\xrightarrow{\lambda^+})^+$ which states that several reductions occur, each of which issue some number of actions λ. These reductions can possibly update the thread store, global state, metadata and identifier components. The latter two occur when a lock comprised a nested lock. See Appendix A for its definition.

  - The thread configuration that is a consequence of executing a lock's constituent commands, T''_j, is a thread configuration where the intermediate lock construct has the state $\mathsf{sblk}(\epsilon)$. That is, it is the thread configuration that precedes the release of a lock's mutex.

  - The final thread configuration T'''_j sees the command that followed the lock being set as the active command.

- Threads in K execute aborting transactions:
  - The transition to T'_k sees the transaction beginning, which updates ld and md, and creates the redo log δ_k.
  - The transaction executes its constituent commands. Recall that the constituent commands of a transaction only update the transaction's redo log – the respective thread's local store and program state are unaffected. Transactional commands can still allocate memory to the free store, however.
  - T''_k is the point at which the transaction wishes to commit.
  - The transaction rolls back in T'''_k so that the active command to execute is the transaction which was just aborted, $\overleftarrow{c}_k$.
- Threads in M go through a similar set of configurations as those in K with the exception that their transactions commit which results in their effect being persisted to their respective thread local store and the program state. A thread that executes a committed transaction sees its active command being set to that which succeeds the transaction.

The predicates labelled D assert the semantics given in Section 4.2.4.2 but for all the threads in each of the thread partitions. The semantics of D have been covered in Section 4.2.4.2. Label E computes the new program state by merging the states of the threads which make positive progress. The merge functions are relatively trivial and we point the reader to Appendix A for their definitions. Note that a data race is informally captured by $\mathsf{MergeStates}$ if there exists two states $\sigma_i \neq \sigma_j$ to be merged such that $v_i \in \mathsf{Dom}(\sigma_i)$, $v_j \in \mathsf{Dom}(\sigma_j)$,

v_i=v_j and $\mathsf{snd}(\sigma_i.\mathsf{s}(v_i)) \neq \mathsf{snd}(\sigma_j.\mathsf{s}(v_j))$, then in σ'=MergeStates($\{\sigma_i, \sigma_j\}$) we have $\mathsf{snd}(\sigma'.\mathsf{s}(v)) = \bot$. That is, the value of v resulting from the merge is undefined. Merging the heaps of the states has a similar semantics. Merging the other components is trivial as their differences are distinct.

The reduction in the conclusion sees the program components being updated to the values constructed in the premise. Also, during the reduction each thread issues a sequence of actions that conforms to Λ defined in Figure 4.3. The syntax $\Lambda_1 \mid\mid \Lambda_2$ denotes that the actions which comprise each sequence may be concurrently interleaved with respect to one another. The only restriction on this interleaving is that the sequence of actions issued by each thread respects their respective thread's program order. The semantics of these actions are covered further in Chapter 5 when we present moverness.

4.3 Summary

In this chapter we have presented a programming model for locks and transactions. Locks are pessimistic, whereas transactions are optimistic. The transactions modelled are out-of-place, weakly isolated and support address based conflict granularity. The semantics of transactions we model are based on the common semantics in leading STM libraries such as [Dice et al., 2006]. A lock and transaction conflict if the transaction accesses the mutex used by a lock. Locks have execution priority over transactions: when a lock and transaction conflict the transaction will always be aborted. This semantics is inline with what the programmer would expect: a lock guarantees run once semantics should it be able to acquire its mutex; by contrast, a transaction always has the potential to

abort. The semantics of objects are those of C structs, which are preserved by transactions. That is, two concurrently executing transactions can freely access distinct fields of the same object and not conflict.

The accesses issued to memory by locks, transactions and commands executed under an uncoordinated semantics are captured at the granularity of actions. An action is roughly analogous to a machine instruction, with the exception that actions focus on capturing: begin and abort/commit of transactions, acquisition/release of mutexes and reads and writes of memory locations. The observation semantics of read actions will be generalised in Chapter 5 when we present *moverness*.

$$\frac{\begin{array}{c}(\underline{\mathsf{UNIFIED{-}NESTED{-}LOCK{-}ACQUIRE}})\\ \ell = \mathsf{VarLocation}(\delta.\mathsf{s}, v)\\ \mathsf{Acquireable}(\ell, \mathsf{md})\\ id' = \mathsf{GenerateID}(\mathsf{md}, \mathsf{Id})\\ \mathsf{md}'{=}\mathsf{md}[id'{\mapsto}(\mathsf{Now}(), \bot, \{\}, \{\}, \{\ell\}, \mathcal{L}(\tau, 1))]\end{array}}{\begin{array}{c}\langle \tau, \mathsf{id}{:}\mathsf{sync}(v)\{c\}, \delta, \mathsf{fs}, \bot, \bot, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle\\ \xrightarrow{\mathsf{ACQ}(\ell)}\\ \langle \tau, id'{:}\mathsf{sblk}(c), \delta, \mathsf{fs}, \bot, \bot, \mathsf{s}_\tau, \sigma, \mathsf{md}', id'\rangle\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{UNIFIED{-}NESTED{-}LOCK{-}ACQUIRE{-}REC}})\\ \ell = \mathsf{VarLocation}(\delta.\mathsf{s}, v)\\ \neg\mathsf{Acquireable}(\ell, \mathsf{md}) \quad \mathsf{HeldByThread}(\tau, \ell, \mathsf{md})\\ \exists id' {\in} \mathsf{Dom}(\mathsf{md}) \;\cdot\; [id'{\mapsto}(\mathsf{beg}, \bot, \gamma_\mathsf{R}, \gamma_\mathsf{W}, \{\ell\}, \mathcal{L}(\tau, count \geq 1))] {\subseteq} \mathsf{md}\\ count'{=}count + 1\\ \mathsf{md}'{=}\mathsf{md}[id'{\mapsto}(\mathsf{beg}, \bot, \gamma_\mathsf{R}, \gamma_\mathsf{W}, \{\ell\}, \mathcal{L}(\tau, count'))]\end{array}}{\begin{array}{c}\langle \tau, \mathsf{id}{:}\mathsf{sync}(v)\{c\}, \delta, \mathsf{fs}, \bot, \bot, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle\\ \xrightarrow{\mathsf{ACQ}(\ell)}\\ \langle \tau, id'{:}\mathsf{sblk}(c), \delta, \mathsf{fs}, \bot, \bot, \mathsf{s}_\tau, \sigma, \mathsf{md}', \mathsf{Id}\rangle\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{UNIFIED{-}NESTED{-}LOCK{-}BLOCKING}})\\ \ell = \mathsf{VarLocation}(\delta.\mathsf{s}, v)\\ \neg\mathsf{Acquireable}(\ell, \mathsf{md})\end{array}}{\begin{array}{c}\langle \tau, \mathsf{id}{:}\mathsf{sync}(v)\{c\}, \delta, \mathsf{fs}, \bot, \bot, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle\\ \xrightarrow{\mathsf{NOP}}\\ \langle \tau, \mathsf{id}{:}\mathsf{sync}(v)\{c\}, \delta, \mathsf{fs}, \bot, \bot, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{UNIFIED{-}NESTED{-}LOCK{-}RELEASE{-}REC}})\\ [id{\mapsto}(\mathsf{beg}, \bot, \gamma_\mathsf{R}, \gamma_\mathsf{W}, \{\ell\}, \mathcal{L}(\tau, count > 1))] {\subseteq} \mathsf{md}\\ count'{=}count - 1\\ \mathsf{md}'{=}\mathsf{md}[id{\mapsto}(\mathsf{beg}, \bot, \gamma_\mathsf{R}, \gamma_\mathsf{W}, \{\ell\}, \mathcal{L}(\tau, count'))]\end{array}}{\begin{array}{c}\langle \tau, id{:}\mathsf{sblk}(\epsilon), \delta, \mathsf{fs}, \bot, \bot, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle\\ \xrightarrow{\mathsf{REL}(\ell)}\\ \langle \tau, \epsilon, \delta, \mathsf{fs}, \bot, \bot, \mathsf{s}_\tau, \sigma, \mathsf{md}', \mathsf{Id}\rangle\end{array}}$$

Figure 4.11: Unified Command Rules (Part I).

$$\frac{\begin{array}{c}(\underline{\mathsf{UNIFIED{-}NESTED{-}LOCK{-}RELEASE}})\\ [id{\mapsto}(\mathsf{beg},\bot,\gamma_\mathsf{R},\gamma_\mathsf{W},\{\ell\},\mathcal{L}(\tau,1))]{\subseteq}\mathsf{md}\\ \mathsf{md}'{=}\mathsf{md}[id{\mapsto}(\mathsf{beg},\mathsf{Now}(),\gamma_\mathsf{R},\gamma_\mathsf{W},\{\},\mathcal{L}(\tau,0))]\end{array}}{\begin{array}{c}\langle\tau,id{:}\mathsf{sblk}(\epsilon),\delta,\mathsf{fs},\bot,\bot,\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle\\ \xrightarrow{\mathsf{REL}(\ell)}\\ \langle\tau,\epsilon,\delta,\mathsf{fs},\bot,\bot,\mathsf{s}_\tau,\sigma,\mathsf{md}',\mathsf{Id}\rangle\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{UNIFIED{-}NESTED{-}LOCK{-}IN}})\\ c\neq\mathsf{id}{:}\mathsf{sync}(\_)\{\_\}\\ \langle\tau,c,\delta,\mathsf{fs},\bot,\bot,\bot,\bot,\bot,\bot\rangle\xrightarrow{\mathsf{REL}(\ell)}\langle\tau,c',\delta',\mathsf{fs}',\bot,\bot,\bot,\bot,\bot,\bot\rangle\\ (\mathsf{s}'_\tau,\sigma')=\mathsf{Persist}(\delta',\mathsf{s}_\tau,\sigma)\end{array}}{\begin{array}{c}\langle\tau,id{:}\mathsf{sblk}(c),\delta,\mathsf{fs},\bot,\bot,\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle\\ \xrightarrow{\lambda^+}\\ \langle\tau,id{:}\mathsf{sblk}(c'),\delta',\mathsf{fs}',\bot,\bot,\mathsf{s}'_\tau,\sigma',\mathsf{md},\mathsf{Id}\rangle\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{UNIFIED{-}NESTED{-}LOCK{-}IN{-}LOCK}})\\ c=\mathsf{id}{:}\mathsf{sync}(\_)\{\_\}\\ \langle\tau,c,\delta,\mathsf{fs},\bot,\bot,\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle\xrightarrow{\lambda^+}\langle\tau,c',\delta',\mathsf{fs}',\bot,\bot,\mathsf{s}'_\tau,\sigma',\mathsf{md}',\mathsf{Id}'\rangle\end{array}}{\begin{array}{c}\langle\tau,id{:}\mathsf{sblk}(c),\delta,\mathsf{fs},\bot,\bot,\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle\\ \xrightarrow{\mathsf{REL}(\ell)}\\ \langle\tau,id{:}\mathsf{sblk}(c'),\delta',\mathsf{fs}',\bot,\bot,\mathsf{s}'_\tau,\sigma',\mathsf{md}',\mathsf{Id}'\rangle\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{UNIFIED{-}ASSIGN}})\\ [v{\mapsto}(\ell_1,val_v),x{\mapsto}(\ell_2,val_x)]{\subseteq}\delta.\mathsf{s}\quad \ell_1{\neq}\ell_2\quad \mathsf{s}'{=}\delta.\mathsf{s}[v{\mapsto}(\ell_1,val_x)]\\ \gamma'_\mathsf{R}{=}\gamma_\mathsf{R}{\cup}\{\ell_2\}\quad \gamma'_\mathsf{W}{=}\gamma_\mathsf{W}{\cup}\{\ell_1\}\quad \delta'{=}(\mathsf{s}',\delta.\mathsf{h})\end{array}}{\langle\tau,v:=x,\delta,\mathsf{fs},\gamma_\mathsf{R},\gamma_\mathsf{W},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle\xrightarrow{\mathsf{R}(\ell_2)\mathsf{W}(\ell_1)}\langle\tau,\epsilon,\delta',\mathsf{fs},\gamma'_\mathsf{R},\gamma'_\mathsf{W},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{UNIFIED{-}FLD{-}UPD}})\\ [v{\mapsto}(\ell_1,\ell_2),x{\mapsto}(\ell_3,val_x)]{\subseteq}\delta.\mathsf{s}\quad \ell_1{\neq}\ell_2\quad \ell_3{\neq}\ell_2\quad \ell_1{\neq}\ell_3\quad \ell_2{\in}\mathsf{Dom}(\delta.\mathsf{h})\\ \ell_f{=}\mathsf{FldLoc}(\delta,v,f)\quad \mathsf{h}'{=}\mathsf{FldUpd}(\delta,v,f,val_x)\quad \delta'{=}(\delta.\mathsf{s},\mathsf{h}')\\ \gamma'_\mathsf{R}{=}\gamma_\mathsf{R}{\cup}\{\ell_1,\ell_3\}\quad \gamma'_\mathsf{W}{=}\gamma_\mathsf{W}{\cup}\{\ell_f\}\end{array}}{\begin{array}{c}\langle\tau,v.f:=x,\delta,\mathsf{fs},\gamma_\mathsf{R},\gamma_\mathsf{W},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle\\ \xrightarrow{\mathsf{R}(\ell_3)\mathsf{R}(\ell_1)\mathsf{W}(\ell_f)}\\ \langle\tau,\epsilon,\delta',\mathsf{fs},\gamma'_\mathsf{R},\gamma'_\mathsf{W},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle\end{array}}$$

Figure 4.12: Unified Command Rules (Part II).

$$\frac{[v{\mapsto}(\ell_1, val_v), x{\mapsto}(\ell_2, \ell_3)]{\subseteq}\delta.\mathsf{s} \quad \ell_1{\neq}\ell_3 \quad \ell_2{\neq}\ell_3 \quad \ell_1{\neq}\ell_2 \quad \ell_3{\in}\mathsf{Dom}(\delta.\mathsf{h}) \quad \ell_f{=}\mathsf{FldLoc}(\delta, v, f) \quad val_f{=}\mathsf{FldVal}(\delta, v, f) \quad \mathsf{s}'{=}\delta.\mathsf{s}[v{\mapsto}(\ell_1, val_f)] \quad \delta'{=}(\mathsf{s}', \delta.\mathsf{h}) \quad \gamma'_{\mathsf{R}}{=}\gamma_{\mathsf{R}}{\cup}\{\ell_2, \ell_f\} \quad \gamma'_{\mathsf{W}}{=}\gamma_{\mathsf{W}}{\cup}\{\ell_1\}}{\langle \tau, v := x.f, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle \xrightarrow{\mathsf{R}(\ell_2)\mathsf{R}(\ell_f)\mathsf{W}(\ell_1)} \langle \tau, \epsilon, \delta', \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma'_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle} \quad (\underline{\mathsf{UNIFIED{-}ASSIGN{-}FLD}})$$

$$\frac{[v{\mapsto}(\ell, val_v)]{\subseteq}\delta.\mathsf{s} \quad (obj, locs){=}\mathsf{CreateObject}(cn, \mathsf{fs}) \quad \mathsf{fs}'{=}\mathsf{fs}{\cup}locs \quad \ell_{\text{base}}{=}\mathsf{Head}(locs) \quad \mathsf{s}'{=}\delta.\mathsf{s}[v{\mapsto}(\ell, \ell_{\text{base}})] \quad \mathsf{h}'{=}\delta.\mathsf{h}[\ell_{\text{base}}{\mapsto}obj] \quad \delta'{=}(\mathsf{s}', \mathsf{h}') \quad \gamma'_{\mathsf{W}}{=}\gamma_{\mathsf{W}}{\cup}\{\ell\}}{\langle \tau, v := \mathsf{new}\ cn, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle \xrightarrow{\mathsf{W}(\ell)} \langle \tau, \epsilon, \delta', \mathsf{fs}', \gamma_{\mathsf{R}}, \gamma'_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle} \quad (\underline{\mathsf{UNIFIED{-}NEW}})$$

$$\frac{[v{\mapsto}(\ell, val_v)]{\subseteq}\delta.\mathsf{s} \quad rslt{=}\mathsf{IsNull}(val_v) \quad \gamma'_{\mathsf{R}}{=}\gamma_{\mathsf{R}}{\cup}\{\ell\}}{\langle \tau, v = \mathsf{null}, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle \xrightarrow{\mathsf{R}(\ell)} \langle \tau, rslt, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle} \quad (\underline{\mathsf{UNIFIED{-}EQ}})$$

$$\frac{[v{\mapsto}(\ell, val_v)]{\subseteq}\delta.\mathsf{s} \quad rslt{=}\neg\mathsf{IsNull}(val_v) \quad \gamma'_{\mathsf{R}}{=}\gamma_{\mathsf{R}}{\cup}\{\ell\}}{\langle \tau, v \neq \mathsf{null}, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle \xrightarrow{\mathsf{R}(\ell)} \langle \tau, rslt, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle} \quad (\underline{\mathsf{UNIFIED{-}NEQ}})$$

$$\frac{\langle \tau, b, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle \xrightarrow{\lambda^+} \langle \tau, b', \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle}{\langle \tau, \mathsf{if}\ b\ \{c1\}\ \mathsf{else}\ \{c2\}, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle \xrightarrow{\lambda^+} \langle \tau, \mathsf{if}\ b'\ \{c1\}\ \mathsf{else}\ \{c2\}, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{ld}\rangle} \quad (\underline{\mathsf{UNIFIED{-}IF}})$$

Figure 4.13: Unified Command Rules (Part IV).

$$\frac{\langle \tau, b, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, \mathsf{True}, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}{\langle \tau, \mathsf{if}\ b\ \{c1\}\ \mathsf{else}\ \{c2\}, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, c1, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}\ (\underline{\mathsf{UNIFIED{-}IF{-}TRUE}})$$

$$\frac{\langle \tau, b, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, \mathsf{False}, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}{\langle \tau, \mathsf{if}\ b\ \{c1\}\ \mathsf{else}\ \{c2\}, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, c2, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}\ (\underline{\mathsf{UNIFIED{-}IF{-}FALSE}})$$

$$\frac{\langle \tau, b, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, b', \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}{\langle \tau, \mathsf{while}\ b\ \{c\}, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, \mathsf{while}\ b'\ \{c\}, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}\ (\underline{\mathsf{UNIFIED{-}WHILE}})$$

$$\frac{\langle \tau, b, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, \mathsf{True}, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}{\langle \tau, \mathsf{while}\ b\ \{c\}, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, c; \mathsf{while}\ b\ \{c\}, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}\ (\underline{\mathsf{UNIFIED{-}WHILE{-}TRUE}})$$

$$\frac{\langle \tau, b, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, \mathsf{False}, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}{\langle \tau, \mathsf{while}\ b\ \{c\}, \delta, \mathsf{fs}, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle \tau, \epsilon, \delta, \mathsf{fs}, \gamma'_{\mathsf{R}}, \gamma_{\mathsf{W}}, \mathsf{s}_\tau, \sigma, \mathsf{md}, \mathsf{Id}\rangle}\ (\underline{\mathsf{UNIFIED{-}WHILE{-}FALSE}})$$

Figure 4.14: Unified Command Rules (Part V).

$$\frac{\begin{array}{c}(\underline{\text{UNIFIED}-\text{METHOD}-\text{CALL}}) \\ [v{\mapsto}(\ell_1,\ell_2)]{\subseteq}\delta.\mathsf{s} \quad \ell_1{\neq}\ell_2 \quad \ell_2{\in}\mathsf{Dom}(\delta.\mathsf{h}) \quad \mathsf{s}{=}\delta.\mathsf{s} \\ c{=}\mathsf{MethodCmds}(\mathsf{TypeOf}(v),m) \quad fargs{=}\mathsf{FormalArgs}(\mathsf{TypeOf}(v),m) \\ (\mathsf{s}_m,locs){=}\mathsf{PassByValue}(\delta,\mathsf{fs},v,p^*,fargs) \\ \mathsf{fs}'{=}\mathsf{fs}{\cup}locs \quad argLocs{=}\mathsf{ArgLocs}(\delta,p^*) \quad \gamma'_{\mathsf{R}}{=}\gamma_{\mathsf{R}}{\cup}\{\ell_1\}{\cup}argLocs \quad \delta'{=}(\mathsf{s}_m,\delta.\mathsf{h})\end{array}}{\begin{array}{c}\langle\tau,v.m(p^*),\delta,\mathsf{fs},\gamma_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle \\ \xrightarrow{\forall\ \ell\ \in\ argLocs\ \cdot\ \mathsf{R}(\ell)} \\ \langle\tau,\mathsf{frame}(c,\mathsf{s}),\delta',\mathsf{fs}',\gamma'_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\text{UNIFIED}-\text{METHOD}-\text{IN}}) \\ \langle\tau,c,\delta,\mathsf{fs},\gamma_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle\tau,c',\delta',\mathsf{fs}',\gamma'_{\mathsf{R}},\gamma'_{\mathsf{W}},\mathsf{s}'_\tau,\sigma',\mathsf{md}',\mathsf{Id}'\rangle\end{array}}{\begin{array}{c}\langle\tau,\mathsf{frame}(c,\mathsf{s}),\delta,\mathsf{fs},\gamma_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle \\ \xrightarrow{\lambda^+} \\ \langle\tau,\mathsf{frame}(c',\mathsf{s}),\delta',\mathsf{fs}',\gamma'_{\mathsf{R}},\gamma'_{\mathsf{W}},\mathsf{s}'_\tau,\sigma',\mathsf{md}',\mathsf{Id}'\rangle\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\text{UNIFIED}-\text{METHOD}-\text{RETURN}}) \\ \delta'{=}(\mathsf{s},\delta.\mathsf{h})\end{array}}{\begin{array}{c}\langle\tau,\mathsf{frame}(\epsilon,\mathsf{s}),\delta,\mathsf{fs},\gamma_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle \\ \xrightarrow{\mathsf{NOP}} \\ \langle\tau,\epsilon,\delta',\mathsf{fs},\gamma_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\text{UNIFIED}-\text{SEQ}-\text{ONE}}) \\ \langle\tau,c_1,\delta,\mathsf{fs},\gamma_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle\tau,c'_1,\delta',\mathsf{fs}',\gamma'_{\mathsf{R}},\gamma'_{\mathsf{W}},\mathsf{s}'_\tau,\sigma',\mathsf{md}',\mathsf{Id}'\rangle\end{array}}{\langle\tau,c_1;c_2,\delta,\mathsf{fs},\gamma_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle\tau,c'_1;c_2,\delta',\mathsf{fs}',\gamma'_{\mathsf{R}},\gamma'_{\mathsf{W}},\mathsf{s}'_\tau,\sigma',\mathsf{md}',\mathsf{Id}'\rangle}$$

$$\frac{\begin{array}{c}(\underline{\text{UNIFIED}-\text{SEQ}-\text{TWO}}) \\ \langle\tau,c_1,\delta,\mathsf{fs},\gamma_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle\tau,\epsilon,\delta',\mathsf{fs}',\gamma'_{\mathsf{R}},\gamma'_{\mathsf{W}},\mathsf{s}'_\tau,\sigma',\mathsf{md}',\mathsf{Id}'\rangle\end{array}}{\langle\tau,c_1;c_2,\delta,\mathsf{fs},\gamma_{\mathsf{R}},\gamma_{\mathsf{W}},\mathsf{s}_\tau,\sigma,\mathsf{md},\mathsf{Id}\rangle \xrightarrow{\lambda^+} \langle\tau,c_2,\delta',\mathsf{fs}',\gamma'_{\mathsf{R}},\gamma'_{\mathsf{W}},\mathsf{s}'_\tau,\sigma',\mathsf{md}',\mathsf{Id}'\rangle}$$

Figure 4.15: Unified Command Rules (Part VI).

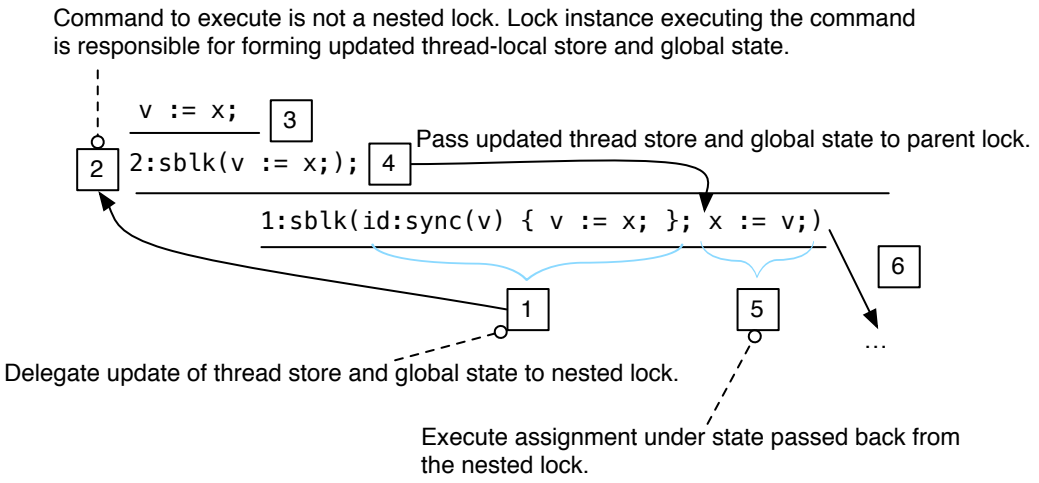


Figure 4.16: Abstract derivation for the delegation of state persistence for nested locks. The responsibility of state persistence is delegated to the most nested lock when executing a lock which is a child of another lock.

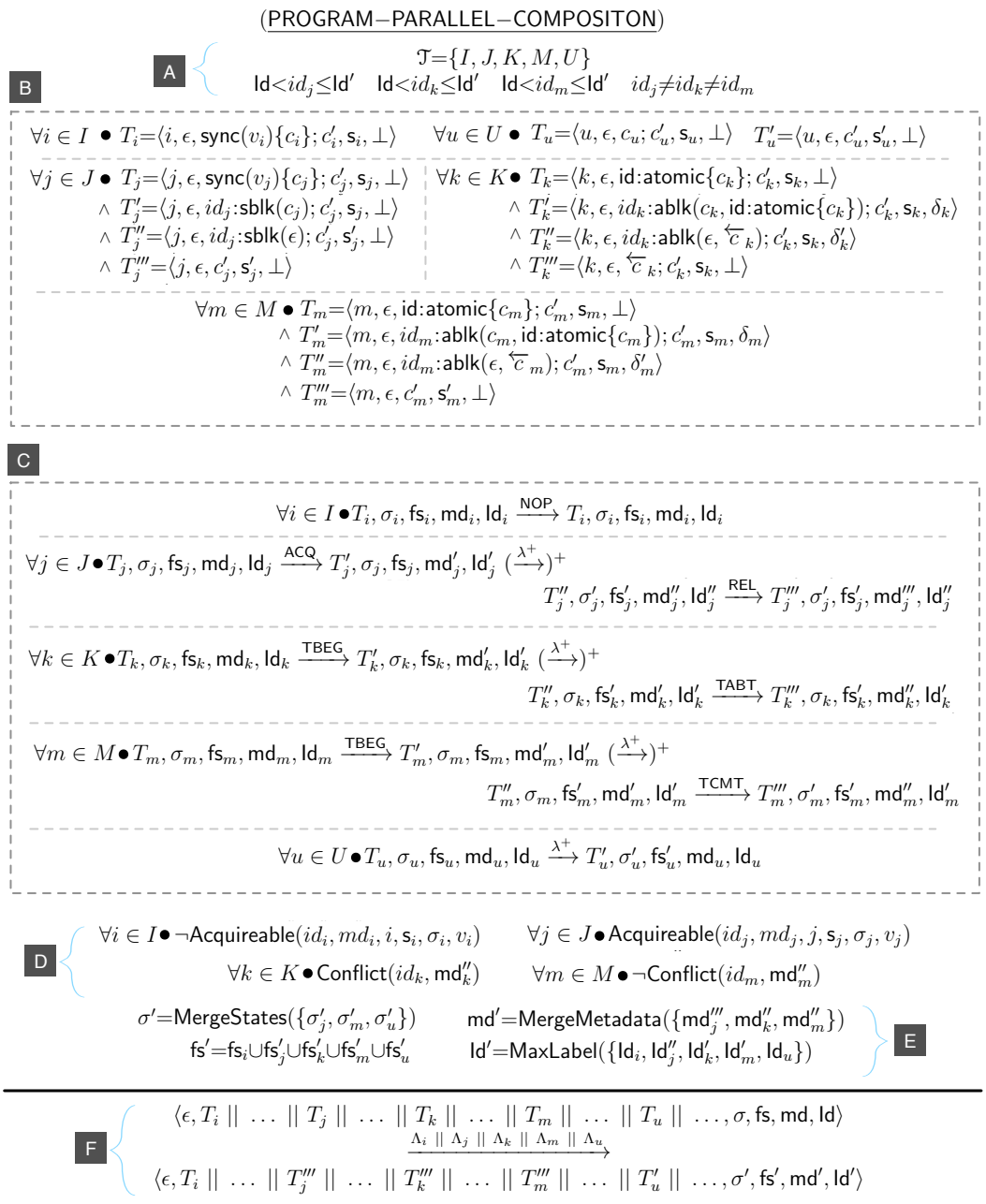


Figure 4.17: Parallel Composition Rule.

Chapter 5

Moverness of Locks and Transactions

5.1 Overview

In Chapter 4 we gave the semantics for locks and transactions. The problem with these semantics is that they require thinking at a low-level of abstraction: determining whether two active transactions conflict requires an understanding of the memory locations they access. Ideally we would reason purely about *observational* properties. That is, if two transactions conflict then the aborted transaction will *observe* the effect of the committed transaction. From the current literature we can find some comparison with memory consistency models. For example, under the Java memory model [Manson et al., 2005] all we really need to know is that if we adhere to the rules, i.e. make appropriate use of `synchronize` and `volatile`, then we are guaranteed certain observational semantics. Observational semantics are far simpler to understand than the mechanics of `synchronized` and `volatile`.

Other memory consistency models such as linearisability [Herlihy and Wing, 1990] define observation guarantees at the granularity of *linearisation points*. That is, the point at which an operation seems to take effect. For example, an `add(int val)` method of a `LinkedList` class may perform the following:

1. Allocate a `Node` object with the value the user provided when invoking `add`.
2. Update the `next` property of the allocated `Node` to be value which the `head` property of the `LinkedList` object currently holds.
3. Update the `head` property of the `LinkedList` object to be that of the allocated `Node` object.

The question here is *when* the effect of `add` is observed to have taken effect. The third step can be considered the linearisation point because it is the phase of `add` which makes the `Node` object allocated by `add` reachable by other clients of the object. Or, more simply, it is the point when we mutate the state of the `LinkedList` object itself. If our `LinkedList` had two properties, `head` and `size`, and the `add` method of `LinkedList` additionally incremented `size`, then the linearisation point would be the point at which the mutation of `head` *and* `size` took place. Under the linearisability memory model we can state that the linearisation point of an operation can take place at *any* point within the operation's execution interval. Most matters in concurrent programming reduce to issues of observation. In languages such as C++, even reasoning about observation semantics in single threaded programs can be non-trivial, as shown in Item 4 of [Meyers, 2005].

The remainder of this chapter looks at applying the general notion of a linearisation point to accesses issued under a lock, transactional and no coordination

semantics. The moves that linearisation points can make with respect to one another are characterised as *free*, *left*, *right* or *both movers*. This *moverness* defines the order that reads and writes take effect and consequently the values that each read may observe. The key benefit of studying the moverness of reads and writes issued in a program that uses locks and transactions is the simplification of an otherwise complex set of observation rules for reads. It has also been shown in previous work [Koskinen et al., 2010] to be of use in purely transactional programming models. The definitions of moverness are based upon the semantics given in Chapter 4.

5.2 Linearisation Points

In this section we give a general intuition of when the linearisation points of commands executed under differing coordination semantics can take place. In Section 5.3 we derive definitions of moverness based on this intuition.

Figure 5.1 shows the notation we use throughout to describe when a linearisation point of a command `c` may take effect. Here, the shaded box below `c` represents its execution interval. The left and right bounds of the interval denote its beginning and completion points. The blue bar denotes the linearisation point for `c` which can be placed at any point within the bounds of `c`'s execution interval. To make our examples simple we assume that all variables are of integer type.

5.2.1 Uncoordinated Commands

Figure 5.2 shows a program where two threads issue uncoordinated accesses to `x`. A total order does not exist over concurrently executing uncoordinated com-

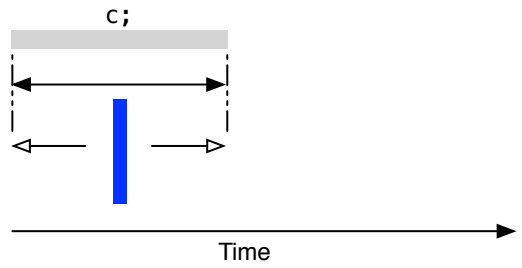


Figure 5.1: The shaded box is the execution interval of `c`. The blue bar (the linearisation point) can be placed at any point within the bounds of `c`'s execution interval.

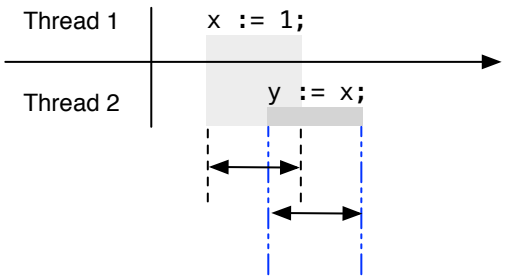


Figure 5.2: The linearisation points of the commands executed by threads 1 and 2 may take place concurrently, resulting in a data race on `x`. This is possible because there does not exist a total ordering over the commands.

mands. That is, the reads and writes issued by each command may take effect concurrently. In Figure 5.2 this is represented by the possibility of each thread's linearisation point occurring concurrently. Concurrent application of the linearisation points of uncoordinated commands does not always lead to erroneous values being observed, as shown in Figure 5.3.

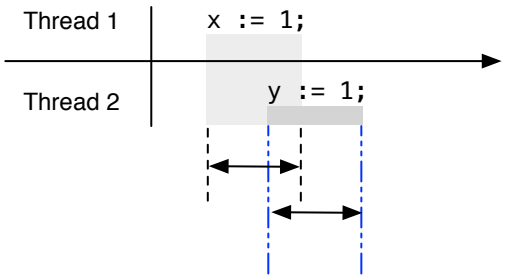


Figure 5.3: The linearisation points of each command can take effect concurrently and not yield erroneous data.

5.2.2 Locks

The linearisation points of concurrently executing locks are totally ordered if and only if they are protected on the same mutex. Consider Figure 5.4. Here, thread 1 acquires `v` and then thread 2 blocks because its lock also wishes to acquire `v`. The linearisation point of thread 2's lock will not take place during the interval of thread 1's lock. Instead, thread 2's linearisation point will occur at some point later. Consequently, thread 2's lock will observe the writes issued by thread 1's lock. That is, thread 2's lock will observe 1 for the value of `x`. Figure 5.5 shows a program where the linearisation points of two locks may take effect concurrently. Here, the value observed by thread 2's read of `x` may be 1, its original value or a junk value due to thread 1 and 2's write and respectively read taking place concurrently. Figure 5.6 gives another example where linearisation points may take place concurrently.

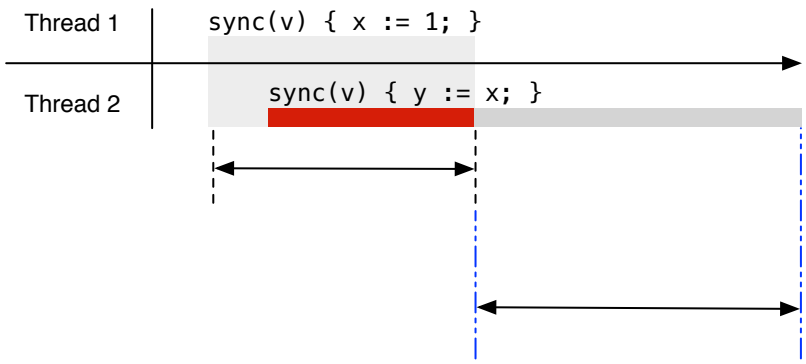


Figure 5.4: Thread 1's lock acquires `v`. Consequently, the linearisation point of thread 2's lock takes place after thread 1's lock.

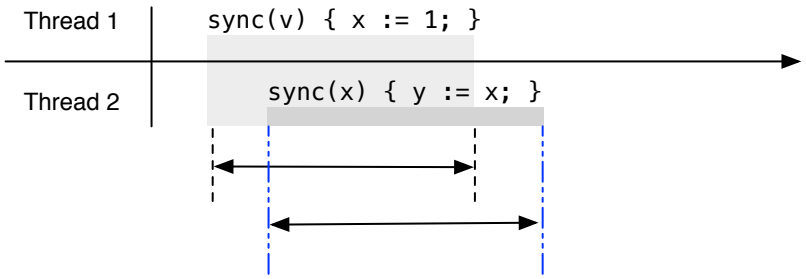


Figure 5.5: Each lock protects its access of `x` on a distinct mutex, consequently a total ordering does not exist over the linearisation points of the locks.

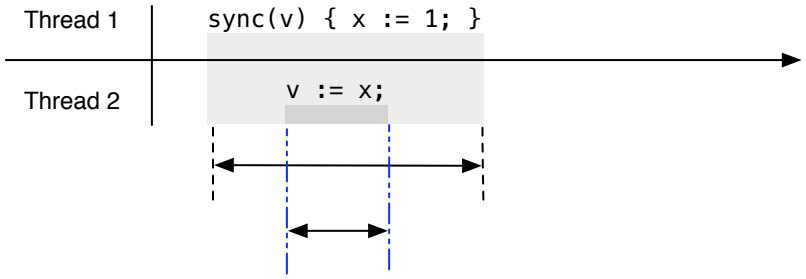


Figure 5.6: The linearisation points may overlap as a total ordering does not exist over the uncoordinated and lock commands.

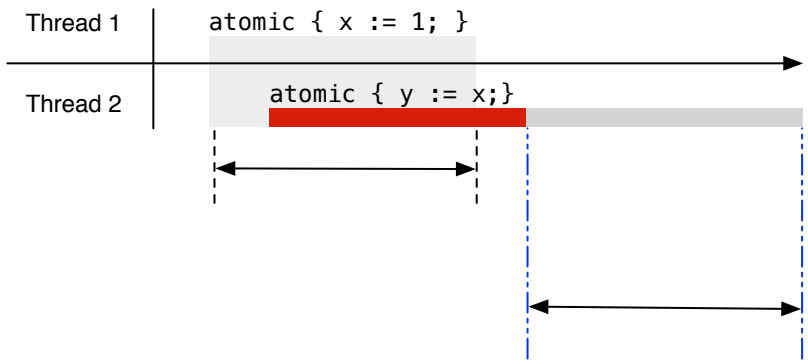


Figure 5.7: The linearisation points of the transactional commands are totally ordered as they conflict. Thread 2's transactional read of `x` will observe 1.

5.2.3 Transactions

The linearisation points of concurrently executing transactions are totally ordered if one transaction writes to memory which the other transaction accesses. For example, in Figure 5.7 thread 2's linearisation point occurs after the linearisation point of thread 1's transaction. A total order does not exist over the linearisation points of transactional accesses to distinct memory, as in Figure 5.8. The linearisation points of transactional and uncoordinated accesses are also not totally ordered as shown in Figure 5.9. Note that the sequence of actions issued by an aborting transaction forms a *ghost sequence*. That is, its removal from any sequence of actions does not affect observational semantics. This is due to transactions in our system being out-of-place. Also note that transactions which abort have no linearisation point; only transactions which commit have a linearisation point.

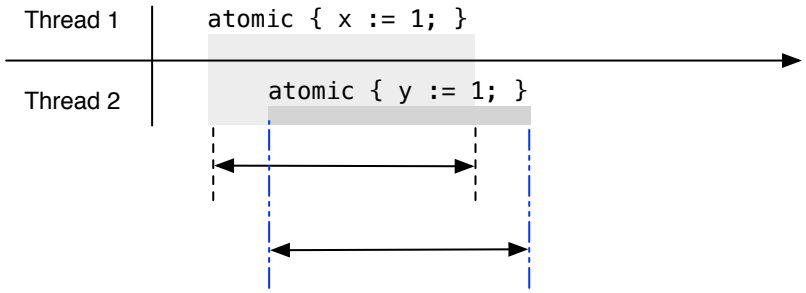


Figure 5.8: A total order does not exist over the linearisation points of transactions which do not conflict.

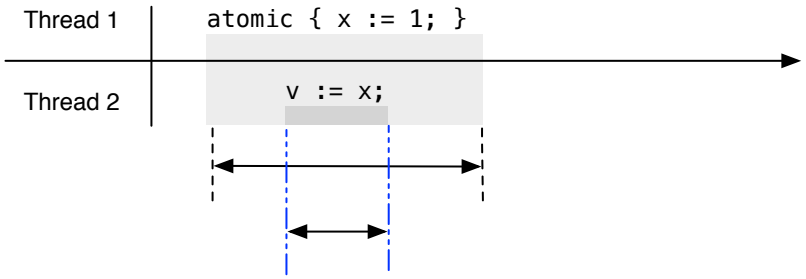


Figure 5.9: The linearisation points of threads 1 and 2 may overlap, resulting in thread 2's read of `x` not observing thread 1's write of `x`.

5.2.4 Locks and Transactions

The linearisation points of a lock and transaction are totally ordered if and only if the transaction accesses the mutex used by the lock. The semantics in Chapter 4 stated that a lock has a stronger semantics than a transaction. That is, if a transaction and lock are executing concurrently, such that the transaction accesses the mutex used by the lock, then the lock will always force the transaction to abort. This is what we mean by a lock having a stronger semantics in the context of concurrently executing locks and transactions. A lock cannot be aborted, but a transaction can. Therefore, the linearisation point of a lock is always ordered before that of a transaction should the previous situation occur, as shown in Figure 5.10. By contrast, Figure 5.11 shows an example of where the linearisation points of the lock and transaction may occur concurrently due to the transaction not accessing the lock's mutex.

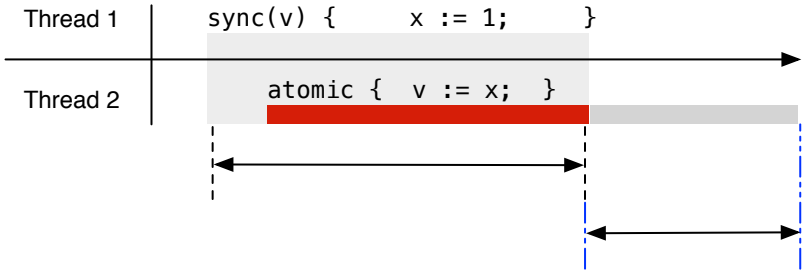


Figure 5.10: The linearisation point of the transaction occurs after that of the lock due to the stronger semantics of locks.

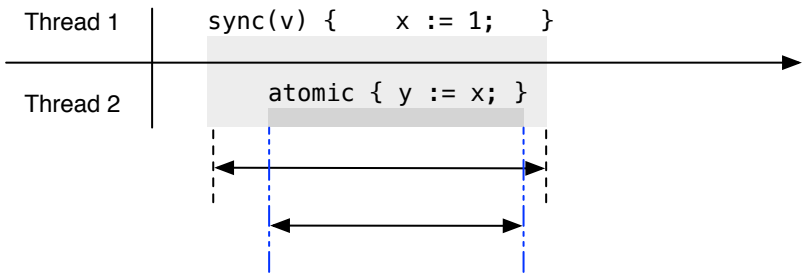


Figure 5.11: The linearisation point of the lock and transaction may occur concurrently due to the transaction not accessing the lock's mutex.

5.3 Moverness

Moverness is a property over sequences of actions which are abstractly represented by linearisation points. For example, if we say that a command c_1 is a left mover with respect to c_2 then we are stating that the actions that c_1 issues take effect before any action issued by c_2, and so on.

Definition 5.1 (Free Mover)**.** Let λ_1^+ be the sequence of actions issued by the command c_1 and λ_2^+ those issued by c_2, such that $c_1 \mid\mid c_2$. The constituent actions of λ_1^+ and λ_2^+ can freely move with respect to one another if and only if:

1. either c_1 or c_2 issue its sequence of actions under an uncoordinated semantics; or

2. c_1 and c_2 issue their respective sequence of actions via locks but protected on distinct mutexes; or

3. c_1 issues its sequence of actions under a lock semantics and c_2 under a transactional semantics, such that c_2's transaction does not access the mutex used by c_1's lock; or

4. c_1 and c_2 issue their respective sequence of actions transactionally, such that c_1 and c_2's transactions do not conflict.

□

Free moving actions may take place in any totally ordered permutation, or concurrently with respect to one another, so long as they respect their issuing thread's program order (Section 2.3.1). The programs given in Figures 5.2, 5.3, 5.5, 5.6, 5.8, 5.9 and 5.11 are free movers.

Example 5.1 (Free Mover – Uncoordinated Commands)**.** Consider the program given in Figure 5.2. The sequence of actions issued by thread 1's command is a write of x, W(x), and the actions issued by thread 2's command is R(x) and W(y). Because the respective actions are free movers with respect to one another the schedule (W(x) || R(x)) W(y) is possible leading to a data race on x. □

Example 5.2 (Free Mover – Non-Conflicting Transactions)**.** Consider the program given in Figure 5.8. The sequence of actions issued by thread 1's transaction is TBEG, W(x) and TCMT, and the sequence of actions issued by thread 2 is TBEG, W(y) and TCMT. The linearisation point of each sequence of actions can take place at any time without introducing a data race. For example, (TBEG W(x) TCMT) || (TBEG W(y) TCMT). □

5.3.1 Left Mover

Definition 5.2 (Left Mover)**.** Let λ_1^+ be the sequence of actions issued by a command c_1 and λ_2^+ be those issued by c_2, such that $c_1 \parallel c_2$. Further, let λ_1^+ be issued under a transactional semantics and λ_2^+ under a lock semantics, such that

there exists a read in λ_1^+ on the mutex used by c_2. We say that the sequence λ_2^+ moves to the left of λ_1^+, λ_2^+ λ_1^+, due to the weaker (abortable) semantics of transactions. That is, the constituent actions of λ_2^+ are guaranteed to take place before any of those in λ_1^+. □

Example 5.3 (Left Mover)**.** Consider the program given in Figure 5.10. The linearisation point of a lock always takes precedent over a transaction when the transaction accesses the mutex used by the lock. Therefore, the only possible sequence of actions initially executed by thread 2 is TBEG R(x) W(v) TABT, with the action sequence ACQ(v) W(x) REL(v) of thread 1's lock moving to the left of thread 2's subsequently issued sequence TBEG R(x) W(v) TCMT. Recall that an aborted transaction has no linearisation point so the actions issued between TBEG and TABT can take place in any total or concurrent order with respect to the constituent actions issued by thread 1's lock. One example of a permissible sequence is ACQ(v) (W(x) || TBEG R(x) W(v)) REL(v) TABT TBEG R(x) W(v) TCMT. Here, thread 1's write of x takes place concurrently with thread 2's transactional read of x and write of v, followed by thread 1 releasing v, thread 2's transaction aborting, and subsequently retrying and committing. The key observation is that the linearisation point of a lock which conflicts with a concurrently executing transaction will always appear to the left of the respective transaction's linearisation point. In the previous example sequence this is represented by all the constituent actions of thread 1's lock being ordered before (or appearing to the left of) the constituent actions of thread 2's committing transactional sequence. For the execution given in Figure 5.10 we may give a stricter claim and state that thread 2's transactional read of x is guaranteed to observe 1. □

5.3.2 Right Mover

Definition 5.3 (Right Mover)**.** A right mover is the mirror of a left mover. Let λ_1^+ be the sequence of actions issued by a command c_1 and λ_2^+ be those issued by c_2, such that $c_1 \mid\mid c_2$. Further, let λ_1^+ be issued under a transactional semantics and λ_2^+ under a lock semantics, such that there exists a read in λ_1^+ on the mutex used by c_2. We say that the sequence λ_1^+ moves to the right of λ_2^+, λ_2^+ λ_1^+, due to the weaker (abortable) semantics of transactions. That is, the constituent actions of λ_2^+ are guaranteed to take place before any of those in λ_1^+. □

The transaction in Figure 5.10 is an example of a right mover.

Example 5.4 (Right Mover)**.** The same as Example 5.3 but interpret "...the linearisation point of a lock which conflicts with a concurrently executing transaction will always appear to the left of the respective transaction's linearisation point." as "...the linearisation point of a transaction which conflicts with a concurrently executing lock will always appear to the right of the respective lock's linearisation point." □

5.3.3 Both Mover

Definition 5.4 (Both Mover)**.** Locks and transactions are both movers with respect to themselves. Let λ_1^+ be the sequence of actions issued by a command c_1 and λ_2^+ be those issued by c_2, such that $c_1 \mid\mid c_2$.

- if λ_1^+ and λ_2^+ are issued under a transactional semantics, and the accesses issued by λ_1^+ and λ_2^+ result in a conflict, then:
  - λ_1^+ can move to the left of λ_2^+, λ_1^+ λ_2^+ (c_1 commits, c_2 aborts); or

– λ_1^+ can move to the right of λ_2^+, λ_2^+ λ_1^+ (c_2 commits, c_1 aborts).

- if λ_1^+ and λ_2^+ are issued under a lock semantics, and the constituent actions of λ_1^+ and λ_2^+ are protected on the same mutex, then:
  - λ_1^+ can move to the left of λ_2^+, λ_1^+ λ_2^+ (c_1 acquires, c_2 blocks); or
  - λ_1^+ can move to the right of λ_2^+, λ_2^+ λ_1^+ (c_2 acquires, c_1 blocks).

□

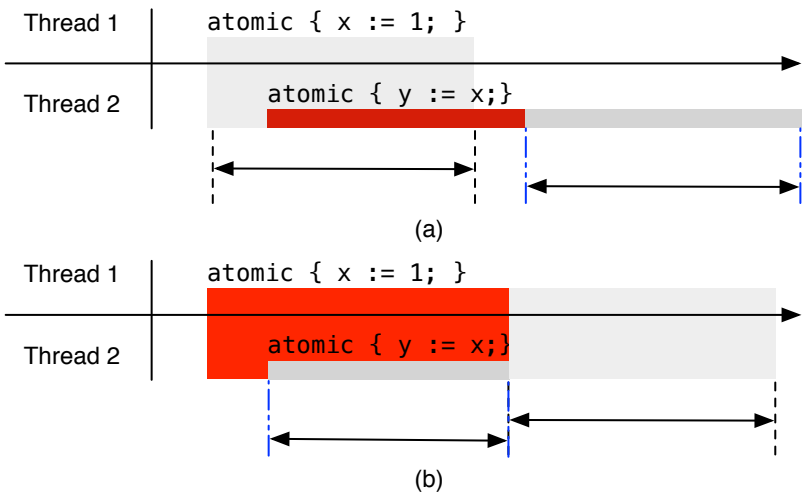


Figure 5.12: (a) The linearisation point of thread 1's transaction appears to the left of the linearisation point of thread 2' transaction. (b) The order of linearisation points is reversed. The order of linearisation points for conflicting transactions is dependent on the contention manager.

Example 5.5 (Both Mover)**.** Consider the program execution given in Figure 5.12. (a) Here, should thread 1's transaction be selected to commit and thread 2's abort, we have for thread 1 TBEG W(x) TCMT and TBEG R(x) W(y) TABT for the initial action sequence of thread 2. Due to the constituent actions of thread

2's first attempt to execute its transaction being ghost actions we have a final sequence that is logically equivalent to TBEG W(x) TCMT TBEG R(x) W(y) TCMT. Now consider the reverse selection for commit/abort as shown in (b). That is, thread 1's transaction initially aborts and thread 2's commits. Here, we have a final sequence equivalent to TBEG R(x) W(y) TCMT TBEG W(x) TCMT. The key observation in this example is that a total ordering exists over the constituent actions of the two transactions but the ordering of each transaction's sequence of actions with respect to one another is dependent upon the contention manager. That is, either thread 2's transaction will observe the actions of thread 1's transaction should thread 2's transaction be selected to abort, or vice versa. □

5.3.4 Moverness and the Java Memory Model

We now show how moverness can be applied to abstract the observation semantics for the happens-before relation in the Java memory model (JMM) Manson et al. [2005].

Under the JMM a program execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$, where

- P is a program;
- A is a set of actions (discussed shortly);
- $\xrightarrow{po}$ is a total ordering over the actions issued by a thread τ;
- $\xrightarrow{so}$ is a total ordering over an execution's synchronisation actions;
- W is a write seen function;
- V is a value written function;

- $\xrightarrow{sw}$ is a partial ordering over synchronisation actions; and
- $\xrightarrow{hb}$ is the transitive closure over $\xrightarrow{po}$ and $\xrightarrow{sw}$.

An action $A = \langle \tau, k, v, u \rangle$, where

- τ is a thread identifier;
- k is the kind of action: read, write, acquire or release;
- v is the variable involved; and
- u is a unique identifier associated with the action.

The write seen function W gives the identifier of the write action a read r observes, e.g. $W(r) = u$. The value written function function W gives the value val written by a write w, e.g. $W(w) = val$. The value observed by a read is a consequence of the preceding write to the same variable in $\xrightarrow{hb}$. The remainder of this section discusses how moverness maps to the JMM Manson et al. [2005] via a series of examples. Note that we only address the happens-before ordering and not the security features of the JMM.

5.3.4.1 Preliminaries

Before proceeding we first establish a connection with the execution environment presented in Chapter 4, particularly when mutexes are acquired and when they are not, determined by the contention manager which resolves accesses to contended memory, irrespective of whether the access issued is transactional or lock-based.

The synchronisation order ($\xrightarrow{so}$) of the JMM is the actual order of acquire/releases taken during a program execution, by contrast to the synchronises-with ($\xrightarrow{sw}$) order which is the relation between release/acquires which may happen. For locks,

both $\xrightarrow{so}$ and $\xrightarrow{sw}$ are straightforward. Transactions are only related should they conflict during an execution, likewise a transaction with a lock. Should a transaction access contended memory then its $\xrightarrow{sw}$ is defined for all the memory it accesses should it conflict with a transaction or lock, with the synchronisation order reflecting the acquire/releases issued. Note that in the cases where a conflict occurs the acquire/releases in the synchronisation order may not actually be required to be executed as the semantics given in Chapter 4 ensures that conflicting transactions/locks are always totally ordered.

5.3.4.2 Examples

Example 5.6 (Conflicting Transactions)**.** Let P be the following:

Thread 1	Thread 2
`atomic {`	`atomic {`
x := y;	z := x;
y := 1;	}
}	

Note that as the two transactions conflict each will acquire/release the mutexes associated with their respective datasets. There are two possible executions (as they are both movers): left or mutually right mover. Note: for conciseness we do not include default release actions on the variables a program accesses.

- *Case 1: Thread 1 commits, Thread 2 aborts.* Let $A = \{\langle 1, ACQ, x, 1\rangle, \langle 1, ACQ, y, 2\rangle, \langle 1, R, y, 3\rangle, \langle 1, W, x, 4\rangle, \langle 1, W, y, 5\rangle, \langle 1, REL, y, 6\rangle, \langle 1, REL, x, 7\rangle, \langle 2, ACQ, z, 8\rangle, \langle 2, ACQ, x, 9\rangle, \langle 2, R, x, 10\rangle, \langle 2, W, x, 11\rangle, \langle 2, REL, x, 12\rangle, \langle 2, REL, z, 13\rangle$, $\xrightarrow{po}$ be the same as the order of each action in A for threads 1 and 2, $\xrightarrow{so}$ be $\langle 1, 7\rangle, \langle 2, 6\rangle, \langle 8, 13\rangle, \langle 9, 12\rangle$, $\xrightarrow{sw}$ be

$\langle 7, 9\rangle, \langle 12, 1\rangle$, $\xrightarrow{hb}$ is as per its definition (in this case, thread 1's actions happen-before any of those issued by thread 2) and W and V be fresh write seen and value seen functions in an execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb}\rangle$. Here, thread 1's transaction can be described as a left mover, thread 2's as a right mover, or (more generally) as being both movers. That is, according to $\xrightarrow{hb}$ the read of x by thread 2 observes the write to x by thread 1.

- *Case 2: Thread 2 commits, Thread 1 aborts.* Let $A = \{\langle 1, ACQ, x, 1\rangle, \langle 1, ACQ, y, 2\rangle, \langle 1, R, y, 3\rangle, \langle 1, W, x, 4\rangle, \langle 1, W, y, 5\rangle, \langle 1, REL, y, 6\rangle, \langle 1, REL, x, 7\rangle, \langle 2, ACQ, z, 8\rangle, \langle 2, ACQ, x, 9\rangle, \langle 2, R, x, 10\rangle, \langle 2, W, x, 11\rangle, \langle 2, REL, x, 12\rangle, \langle 2, REL, z, 13\rangle$, $\xrightarrow{po}$ be the same as the order of each action in A for threads 1 and 2, $\xrightarrow{so}$ be $\langle 8, 13\rangle, \langle 9, 12\rangle, \langle 1, 7\rangle, \langle 2, 6\rangle$, $\xrightarrow{sw}$ be $\langle 7, 9\rangle, \langle 12, 1\rangle$, $\xrightarrow{hb}$ is as per its definition (in this case, thread 2's actions happen-before any of those issued by thread 1) and W and V be fresh write seen and value seen functions in an execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb}\rangle$. Here, thread 2's transaction can be described as a left mover and thread 1's as a right mover. That is, according to $\xrightarrow{hb}$ the read of x by thread 2 observes the original value of x. □

Example 5.7 (Non-Conflicting Transactions)**.** Let P be the following:

Thread 1	Thread 2
`atomic {`	`atomic {`
x := y;	z := 1;
y := 1;	}
}	

Let $A = \{\langle 1, R, y, 1\rangle, \langle 1, W, x, 2\rangle, \langle 1, W, y, 3\rangle, \langle 2, W, z, 4\rangle\}$, $\xrightarrow{po}$ be the same as the order of each action in A for threads 1 and 2, $\xrightarrow{so}$ and $\xrightarrow{sw}$ (with the exception of the initial releases injected on the variables) be empty, $\xrightarrow{hb}$ is as per its definition and W and V be fresh write seen and value seen functions in an execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb}\rangle$. Consequently, the actions issued by threads 1 and 2 are free movers. That is, the accesses issued by each thread are unrelated in $\xrightarrow{hb}$. □

Example 5.8 (Conflicting Lock and Transaction)**.** Let P be the following:

Thread 1	Thread 2
`atomic {`	`sync(x) {`
x := y;	z := 1;
y := 1;	}
}	

Note that to make the connection in the JMM we must conservatively issued acquire/releases on the transactions dataset due to the conflict with the lock. Let $A = \{\langle 2, ACQ, x, 1\rangle, \langle 1, W, z, 2\rangle, \langle 2, REL, x, 3\rangle, \langle 1, ACQ, x, 4\rangle, \langle 1, ACQ, y, 5\rangle, \langle 1, R, y, 6\rangle, \langle 1, W, x, 7\rangle, \langle 1, W, y, 8\rangle, \langle 1, REL, y, 9\rangle, \langle 1, REL, x, 10\rangle\}$, $\xrightarrow{po}$ be the same as the order of each action in A for threads 1 and 2, $\xrightarrow{so}$ be $\langle 1, 3\rangle, \langle 4, 10\rangle, \langle 5, 9\rangle$, $\xrightarrow{sw}$ be $\langle 3, 4\rangle, \langle 10, 1\rangle$, $\xrightarrow{hb}$ is as per its definition and W and V be fresh write seen and value seen functions in an execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb}\rangle$. Consequently, thread 2 is a left mover w.r.t. thread 1. That is, the actions issued by thread 2 will happen-before those issued by thread 1 as a lock is a left mover w.r.t. a transaction when the transaction accesses the mutex the lock is protected on. □

Example 5.9 (Non-Conflicting Lock and Transaction)**.** Let P be the following:

Thread 1	Thread 2
`atomic {`	`sync(z) {`
x := y;	z := 1;
y := 1;	}
}	

Let $A = \{\langle 1, R, y, 1\rangle, \langle 1, W, x, 2\rangle, \langle 1, W, y, 3\rangle, \langle 2, ACQ, z, 4\rangle, \langle 2, W, z, 5\rangle, \langle 2, REL, z, 6\rangle\}$, $\xrightarrow{po}$ be the same as the order of each action in A for threads 1 and 2, $\xrightarrow{so}$ is $\langle 4, 6\rangle$, $\xrightarrow{sw}$ contains only a relationship between the initial release injected on z and the acquire performed in the program text, $\xrightarrow{hb}$ is as per its definition and W and V be fresh write seen and value seen functions in an execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb}\rangle$. Consequently, the actions issued by threads 1 and 2 are free movers. That is, the actions issued by threads 1 and 2 are unrelated in $\xrightarrow{hb}$. □

5.4 Summary

In this chapter we have presented moverness for accesses issued under a lock, transactional and no coordination semantics. Moverness abstracts the underlying machine's semantics for these coordination semantics and defines them as observation rules. Moverness can be seen as a being a memory consistency model [Adve and Gharachorloo, 1996] with the respective move definitions defining the observational properties of reads. Indeed, we showed this by mapping its abstract semantics to the lower-level execution semantics of the Java memory model. Its key benefit is that it simplifies an otherwise complex set of observation rules for reads issued by a program using both locks and transactions, as shown in Section

5.3.4. Each coordination type is associated a linearisation point [Herlihy and Wing, 1990] and a priority. The movement of one linearisation point with respect to another linearisation point falls under the semantics of a free, left, right or both mover. The linearisation points of unrelated accesses are free movers. A lock has execution priority of a transaction and is classified to be a left mover, by contrast to a transaction which is a right mover. Timing events result in the requirement for defining both mover semantics: a lock can move to the left or right of another lock protected on the same mutex, depending on which lock acquired the mutex first; likewise, a transaction that conflicts with another transaction can move to the left or right of the other transaction depending on the commit/abort selection of the contention manager.

Chapter 6

Guaranteed Transactions

6.1 Overview

Locks and transactions are tools used to serialise accesses to memory. The level of complexity required to serialise accesses to memory under locks is significantly greater than that of transactions. However, there are cases when transactional semantics, specifically those under a weakly isolated STM, are insufficient for executing certain types of operation. Such operations include I/O, CPU bound tasks and any general form of irreversible operation. In these cases the programmer must apply locks or the privatisation/publication idioms. It can be argued that neither approach is ideal in a transactional program:

Locks require the programmer to explicitly maintain lock invariants, e.g. mutexes, read/write locks, etc. This process is often error prone due to orderings over lock invariants being hard to track and enforce, particularly in object oriented systems where components are frequently composed. Additionally, while being the most practical, mixing locks and transactions can

result in a significantly more complex programming model. (Part II presents a framework for determining the data-race-freedom of such a programming model, which we found to justify the previous claim.)

Privatisation/Publication requires the programmer to explicitly maintain reachability of the object graph. This is no less error prone than the use of locks. It is arguable as to which is the more challenging: managing the reachability of a program's object graph, or maintaining lock invariants. Nonetheless, the principal advantage of the privatisation/publication idioms is that it permits the programmer to stay within a transactional programming model. That is, the programmer can rely completely on application of transactional semantics.

On a more idealistic level both locks and the privatisation/publication idioms are inappropriate because they go against the original philosophy of STM [Shavit and Touitou, 1995]. The goal of STM was to significantly lower the entry bar for creating *correct*, i.e. data-race-free, concurrent programs. That is, make it hard to get it wrong and easy to get it right. Combining transactions and locks/the privatisation/publication idioms leaves the programmer in an awkward and often complex environment when, on occasion, he/she requires a stronger coordination semantics.

Guaranteed transactions attempt to address the deficiencies of locks and the privatisation/publication idioms. A guaranteed transaction is a means to get a stronger coordination semantics but via an abstraction akin to transactions. That is, when, on occasion, the programmer requires a stronger semantics he can transparently substitute the `atomic` keyword for `gatomic`, the keyword we asso-

ciate with a guaranteed transaction. The programmer does not have to maintain isolation invariants or worry about object graph reachability issues. Guaranteed transactions, like transactions, can be thought of as being similar to garbage collection in an environment like the Java Virtual Machine (JVM): the programmer is not required to understand how the JVM's garbage collector works or what algorithm it uses, he only needs to be aware of the fact that the system will ensure that unreachable memory will be freed. Briefly, the benefits of guaranteed transactions over locks and the privatisation/publication idioms are as follows:

- Abstract parity with transactions. Mixing transactions and guaranteed transactions is a transparent process.
- Implicit handling of isolation and object graph reachability invariants.
- Inherently support a concurrency model that is similar to read/write locks. That is, if possible, several guaranteed transactions can execute concurrently even if the datasets of the guaranteed transactions intersect.

The remainder of this section briefly recaps the problems associated with the use of locks and the privatisation/publication idioms in a transactional program, followed by an overview of guaranteed transactions.

6.1.1 Locks

Locks require the programmer maintain isolation invariants. These isolation invariants are typically come in the form of a mutex or a more specific form such as a read/write lock. Locks, when applied in a routine and consistent manner, can be used to support most types of serialisation semantics. Nonetheless, the

application of locks, particularly for fine grained concurrency control, has a steep learning curve which can take several years to master. Figure 6.1 shows a coarse and fine grained locking strategy. Here, (a) protects all accesses to `x`, `y` and `z` on a single mutex. The advantage of the coarse grained approach is that serialising accesses to defined regions of memory is simpler as we have less mutexes to juggle before performing the appropriate accesses. However, using a coarse grained approach significantly reduces the amount of concurrency that may be exploited. For example, one thread may only access `x` but still have to contend for the same lock that another thread which accesses `v` and `y` requires. The fine grained approach as shown in (b) is the opposite: we use a mutex associated with each variable. The advantage of this approach is that a thread which only accesses `x` does not need to contend with another thread that accesses `v` and `y`. However, as Figure 6.1 shows, even for a trivial program the use of fine grained locks can become quite complex, and lead to the likes of deadlocks should a consistent ordering not be maintained over lock acquisitions. For example, in (b) we acquire locks in a lexicographic ordering to prevent deadlock.

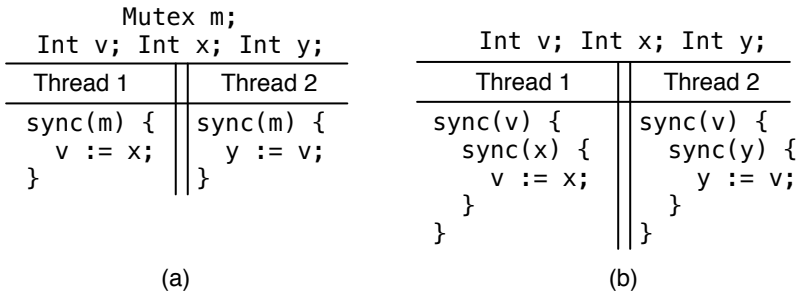


Figure 6.1: (a) A single mutex is used to protect accesses to `x`, `y` and `z`. (b) The individual mutexes associated with `x`, `y` and `z` are used to protect their respective accesses.

We can extend fine grained locking further by associating each variable with a read/write lock. Under this approach multiple readers can execute concurrently but multiple writers cannot. Fine grained read/write locking is tricky to apply but significantly reduces lock contention while not prohibiting concurrency with respect to read operations. Figure 6.2 shows an example program that uses a fine grained read/write lock strategy. Thread 1 acquires the write lock associated with `v`, `v_rw`, as thread 1 wishes to write `v`. Thread 1 only needs to acquire `x`'s associated read lock as thread 1 only reads `x`, as does thread 2 which additionally acquires the write lock associated with `y`. Threads 1 and 2 can execute their operations concurrently. Multiple threads can acquire the read lock of a read/write lock, but only one thread can have acquired the write portion of a lock at any given time.

```
ReadWriteLock v_rw;
ReadWriteLock x_rw;
ReadWriteLock y_rw;
Int v; Int x; Int y;
```

Thread 1	Thread 2

`sync(v.WriteLock) {`
`  sync(x.ReadLock) {`
`  v := x;`
`  }`
`}`
	`sync(x.ReadLock) {`
	`  sync(y.WriteLock) {`
	`  y := x;`
	`  }`
	`}`

Figure 6.2: Each variable has an associated read/write lock.

As described previously a transactional program that uses a weakly isolated STM must use either locks or the privatisation/publication idioms when wishing to execute an irreversible operation. It is also possible that a programmer may want to use one of the two previous alternatives when executing a CPU task, or when the performance budget of a program leaves little margin for possible retries

of transactions which abort. Using locks and transactions in the same program is non-trivial. There are two main issues:

- Management of lock invariants. The complexity of this task depends largely on the locking strategy a code base uses – fine or coarse grained, etc.
- Isolation of accesses to memory issued by locks and transactions. The programmer must understand when a lock and transaction "conflict" and the semantics of such a conflict.

The use of locks in a transactional program naturally makes coordination of accesses more complex. For example, consider Figure 6.3. Here, there is no particular reason to use a lock to coordinate thread 1's write and read of `v` and respectively `x`. The point of Figure 6.3 is that it is not immediately obvious, even for such a trivial program, whether or not there exists a total ordering over thread 1 and 2's respective accesses of `v`. In this case the accesses issued to `v` by each thread are serialised as thread 2's transaction accesses the mutex used by thread 1's lock. The point here is that mixing locks and transactions increases the complexity of the programming model significantly, but affords the programmer more powerful options for coordinating accesses. Furthermore, as discussed previously, the thesis of transactional memory was to reduce the learning curve for writing obviously correct concurrent programs. The use of locks within a transactional program re-introduces the steep learning curve that STM intended to dispose of.

To motivate the need of a stronger semantics consider Figure 6.4. Here, should the thread executing the write to disk abort, the write to disk will still persist. The atomicity, consistency and isolation guarantees of transactions only hold for in-memory data. In this case disk is excluded from such guarantees. Therefore,

```
Int v; Int x; Int y;
```

Thread 1	Thread 2

`sync(v) {`
`  v := x;`
`}`
	`atomic {`
	`  y := v;`
	`}`

Figure 6.3: Using locks and transactions.

it is possible that other transactions may observe the value written by an aborted transaction. Locks can be used to remedy this situation as shown in Figure 6.5. Here, the write of disk will not execute more than once. However, the problem remains that introducing a lock (or most likely several locks) into a transactional program removes the intuitiveness of a purely transactional program. The more locks that are required in a transactional program, the less appealing transactions become.

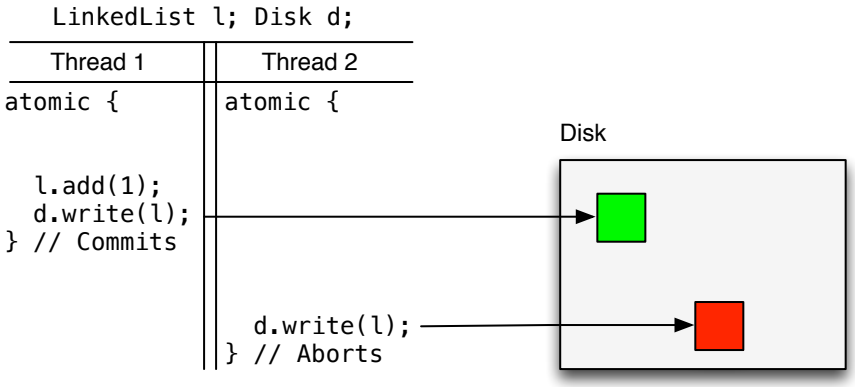


Figure 6.4: Using transactions to execute an irreversible I/O operation. Thread 2's transaction aborts but its write to disk remains. Thread 2's transaction has invalidated the atomicity and consistency guarantees.

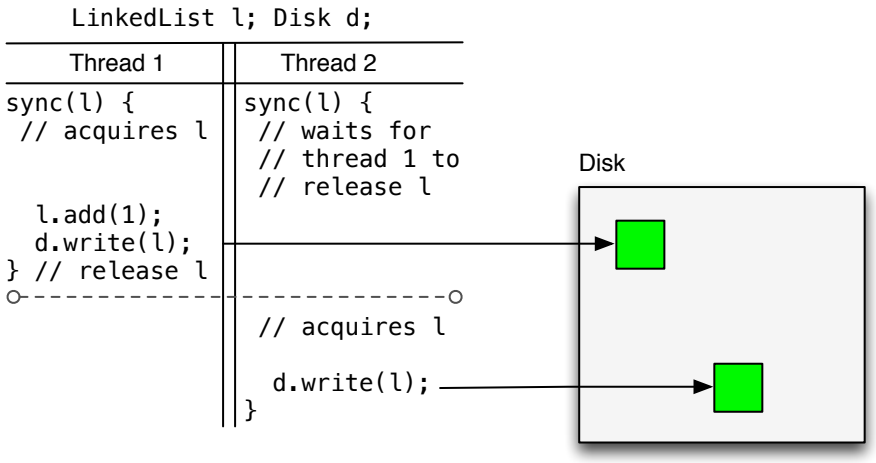


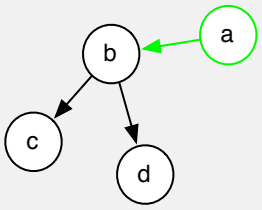
Figure 6.5: Using locks to safely execute an irreversible I/O operation.

6.1.2 Privatisation/Publication Idioms

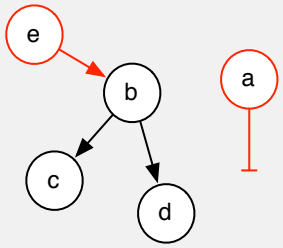
The privatisation and publication idioms [Spear et al., 2007] provide a means for dropping in and out of a strong semantics without the use of locks or other forms of coordination control. The general principle of the idioms is shown in Figure 6.6. To draw comparison to locks, which use mutexes, etc. to encode isolation invariants, the privatisation/publication idioms use standard program logic to control the reachability of an object (or objects). Controlling the reachability of an object requires the programmer view his program as one large graph, the program's *object graph*. During the execution of a program objects become reachable from one another be performing assignments. For example, executing `o.f := v` results in the memory `v` referencing being reachable by `o`. In graph parlance execution of the previous command results in a directed edge labelled `f` from a node labelled `o` to a node labelled `v`. It is not uncommon for an object graph of even a basic program to entail thousands of nodes, particularly in object heavy

languages such as Java and (more so) Ruby [Flanagan and Matsumoto, 2008].

```
// b and its subgraph can be reached by
// multiple threads through a
```




```
atomic {
  // Privatise.
  // cut off connectivity to
  // b by other threads
  // introduce thread-local connection
  // to b's subgraph
}
```




```
// operate on b's subgraph; no coordination required
ComplexOperation(b);
```

```
atomic {
  // Publicise.
  // make b reachable again
}
```

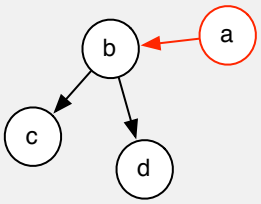


Figure 6.6: General principle of the privatisation and publication idioms. Transactions are used to close off and open up the reachability of a program's object graph.

Figure 6.7 gives a simplified version of applying the privatisation and publication idioms for writing the contents of a linked list to disk. The program attempts to replicate the semantics of Figure 6.5. Here, we use the first transaction to set a thread local variable `l1` to point-to the first node of the linked list that `l` points-to. Subsequently, we close off (privatise) the reachability to the nodes that `l1`

points-to by setting `l` to null. Observe that `l` is accessible by all threads but `l1` is only accessible by the executing thread. Only the executing thread can now access the nodes of the linked list that `l1` points-to, so we execute the irreversible `write` operation. Our final step is to open up (publicise) the nodes that `l1` refers to so that all threads may observe the list. We do this by updating `l` to point-to the memory that `l1` points-to.

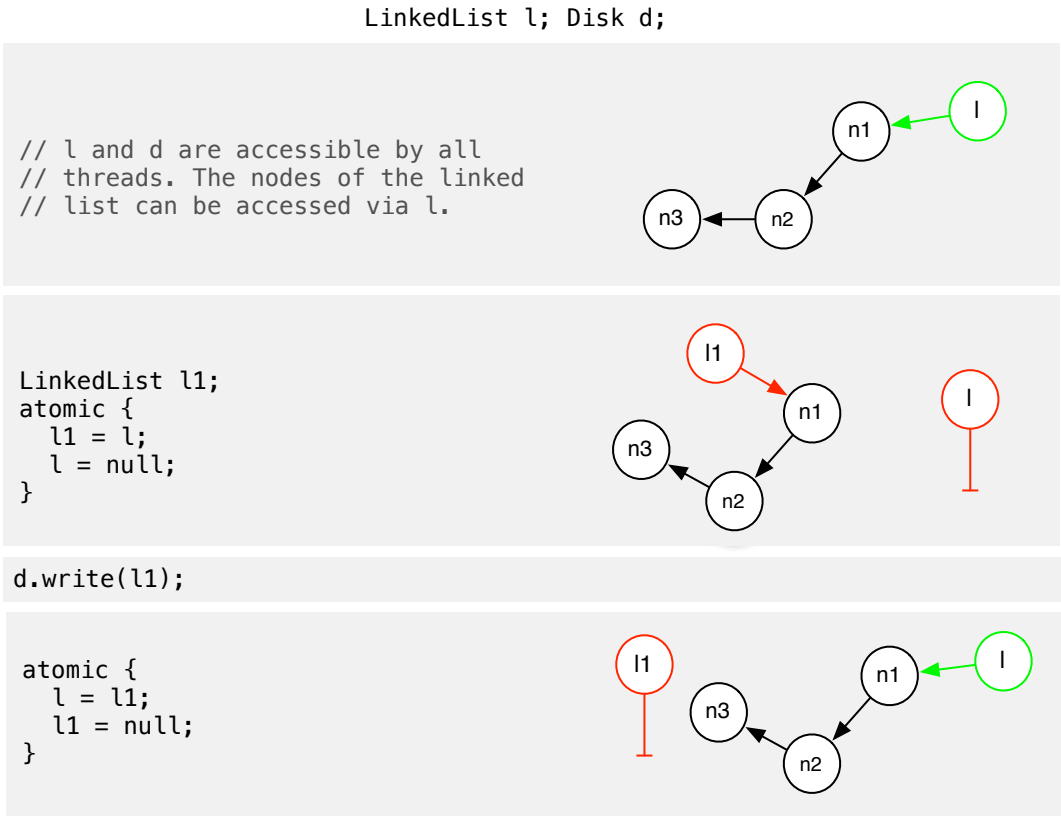


Figure 6.7: Simplified application of the privatisation/publication idioms to write a linked list's contents to disk.

6.1.3 Guaranteed Transactions

A guaranteed transaction is another coordination tool that is designed to keep the programmer in the transactional world for as long as possible. A guaranteed transaction complements transactions and locks. We specifically position a guaranteed transaction as a means to safely (remove the programmer from manually maintaining isolation invariants) use the privatisation/publication idioms and supplant usages of locks in specific scenarios. For example, a guaranteed transaction is ideal to simplify the semantics of the program given in Figure 6.5. The rest of this section outlines some of the benefits of guaranteed transactions and also positions it with respect to the current literature.

A guaranteed transaction lies between the semantics of a transaction and lock. That is, it provides a stronger semantics than a transaction but affords a less precise semantics to custom rolled locking strategies. Guaranteed transactions are pessimistic like locks. That is, the environment must be in a state to satisfy the invariants (read and write set) before a guaranteed transaction can begin execution. Guaranteed transactions are a good solution when the data to be privatised is relatively small and the object graph is predictable, e.g. acyclic. Candidate data structures include the likes of linked lists and trees. Guaranteed transactions are not free: isolation invariants are computed at runtime and can abort actively running transactions which conflict with such computation. Consequently, guaranteed transactions should be used when the data to be privatised is not heavily contended and has a simple object graph. Using guaranteed transactions to privatise heavily contented data or to privatise objects with large object graphs will most likely result in increasing the amount of memory contention.

Guaranteed transactions are similar to *obstinate* transactions [Ni et al., 2008] but are not a product of a prior abort. [Welc et al., 2008] use single owner read locks to transition to a guaranteed semantics but permit only a single such semantics to run at any given time. Multiple guaranteed transactions can execute concurrently provided they do not conflict. [Sonmez et al., 2009] present a model built on Haskell STM that turns atomics that access "hot" regions of memory into pessimistic atomics, however this approach again is dynamic and does not afford dataset guarantees. Recent literature such as that by [McCloskey et al., 2006; Ni et al., 2008; Shavit and Matveev, 2012] and [Welc et al., 2008] have, via empirical evidence, justified not only the practical feasibility of pessimistic concurrency control for STM but also its importance in simplifying the programming model.

An example application of a guaranteed transaction is shown in Figure 6.8. Here, the guaranteed transaction and transaction conflict. Should they be scheduled concurrently the guaranteed transaction will always commit and force the conflicting transaction (and any other conflicting transactions which execute during the guaranteed transaction's interval) to abort. Observe that a guaranteed transaction does not require the programmer specify any invariants. The simplicity of guaranteed transactions comes at the cost of over approximating its dataset. For Figure 6.8 a guaranteed transaction is ideal as the object graph of the guaranteed transaction is simple.

Guaranteed transactions can execute concurrently if their respective datasets do not conflict, as shown in Figure 6.9. Here, both guaranteed transactions only read the contents of the linked list pointed-to by `l`. Consequently, they can be executed concurrently. This is a slightly contrived example only used for illustration – thread 2's invocation of `l.traverse()` would ideally use the more

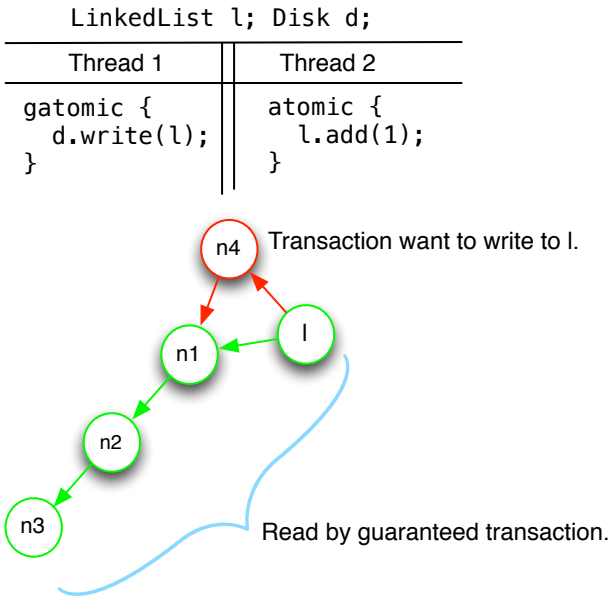


Figure 6.8: The guaranteed transaction reads the memory associated with `l`,`n1`, `n2` and `n3`. `l` is included in the transaction executed by thread 2's write set. The guaranteed transaction will force the transaction to abort should they be scheduled concurrently.

efficient semantics of a transaction as the guaranteed transaction in this instance is not required. The guiding philosophy of guaranteed transactions are loosely based upon a quote by Simon Peyton-Jones from a talk he gave in 2006[1] on the topic of STM, paraphrased: "...would you rather a fast program that is correct some of the time or a slower program that is correct all of the time?" This quote has resonated with me deeply when thinking about coordination in non-trivial programs. Conflicting guaranteed transactions are totally ordered, as shown in Figure 6.10.

[1]Developer!Developer!Developer! conference held at Microsoft's Campus in Reading, UK. At the time I was an intern at Microsoft.

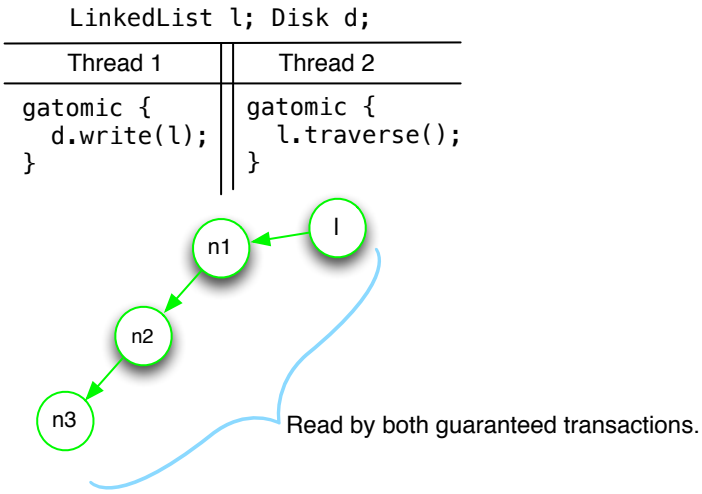


Figure 6.9: The guaranteed transactions can execute concurrently as neither guaranteed transaction writes data the other guaranteed transaction accesses.

The remainder of this chapter is structured as follows:

- Section 6.2 gives the operational semantics of guaranteed transactions. This includes thread defined commands and a modified version of the parallel composition rule given in Section 4.2.5.
- Section 6.3 defines the moverness (Chapter 5) of guaranteed transactions within a transactional program.

6.2 Rules

Before we present the rules for guaranteed transactions we redefine the definition of Coord to be Coord $\stackrel{\text{def}}{=} \mathcal{A} \mid \mathcal{G}$, where $\mathcal{A}$ is a transaction and $\mathcal{G}$ is a guaranteed transaction. Just as in Chapter 4 we use the values of Coord to distinguish the

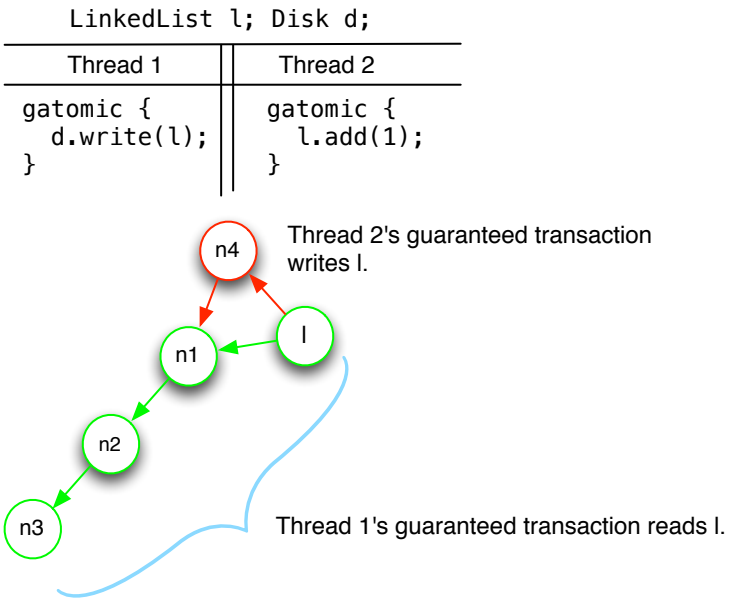


Figure 6.10: Conflicting guaranteed transactions are totally ordered should they be scheduled concurrently.

coordination semantics that the metadata in md models. We also extend the definition of λ and Λ, originally defined in Figure 4.3, to be $\lambda \stackrel{\text{def}}{=} \ldots \mid \mathsf{GBEG} \mid \mathsf{GCMT}$ and respectively $\Lambda \stackrel{\text{def}}{=} \ldots \mid \mathsf{GBEG}\ \lambda^{+}_{RW}\ \mathsf{GCMT}$. Note also that guaranteed transactions, like transactions, are flattened and are associated with a unique label. Transactions and guaranteed transactions can be mutually nested but are flattened, like nested transactions.

6.2.1 Thread Rules

The thread command rules for guaranteed transactions are given in Figure 6.11.

(THREAD−GTRANSACTION−BEGIN) begins execution of a guaranteed transaction:

$$\frac{\begin{array}{c}(\underline{\mathsf{THREAD{-}GTRANSACTION{-}BEGIN}})\\ reads{=}\mathsf{Reads}(c,\mathsf{s}_\tau,\sigma)\quad writes{=}\mathsf{Writes}(c,\mathsf{s}_\tau,\sigma)\\ \neg\mathsf{GConflict}(writes,\mathsf{md})\\ id'{=}\mathsf{GenerateID}(\mathsf{md},\mathsf{Id})\\ \mathsf{md}'{=}\mathsf{md}[id'{\mapsto}(\mathsf{Now}(),\bot,reads,writes,reads{\cup}writes,\mathcal{G})]\end{array}}{\begin{array}{c}\langle\tau,\epsilon,\mathsf{id}{:}\mathsf{gatomic}\{c\},\mathsf{s}_\tau,\bot\rangle,\sigma,\mathsf{fs},\mathsf{md},\mathsf{Id}\\ \xrightarrow{\mathsf{GBEG}}\\ \langle\tau,\epsilon,id'{:}\mathsf{gablk}(c),\mathsf{s}_\tau,\bot\rangle,\sigma,\mathsf{fs},\mathsf{md}',id'\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{THREAD{-}GTRANSACTION{-}BLOCK}})\\ writes{=}\mathsf{Writes}(c,\mathsf{s}_\tau,\sigma)\quad \mathsf{GConflict}(writes,\mathsf{md})\end{array}}{\begin{array}{c}\langle\tau,\epsilon,\mathsf{id}{:}\mathsf{gatomic}\{c\},\mathsf{s}_\tau,\bot\rangle,\sigma,\mathsf{fs},\mathsf{md},\mathsf{Id}\\ \xrightarrow{\mathsf{NOP}}\\ \langle\tau,\epsilon,\mathsf{id}{:}\mathsf{gatomic}\{c\},\mathsf{s}_\tau,\bot\rangle,\sigma,\mathsf{fs},\mathsf{md},\mathsf{Id}\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{THREAD{-}GTRANSACTION{-}IN}})\\ \delta{=}(\mathsf{s}_\tau{\cup}\sigma.\mathsf{s},\sigma.\mathsf{h})\\ \langle\tau,c,\delta,\mathsf{fs},\bot,\bot,\bot,\bot,\bot,\bot\rangle\xrightarrow{\lambda^+}\langle\tau,c',\delta',\mathsf{fs}',\bot,\bot,\bot,\bot,\bot,\bot\rangle\\ (\mathsf{s}'_\tau,\sigma'){=}\mathsf{Persist}(\delta',\mathsf{s}_\tau,\sigma)\end{array}}{\begin{array}{c}\langle\tau,\epsilon,id{:}\mathsf{gablk}(c),\mathsf{s}_\tau,\bot\rangle,\sigma,\mathsf{fs},\mathsf{md},\mathsf{Id}\\ \xrightarrow{\lambda^+}\\ \langle\tau,\epsilon,id{:}\mathsf{gablk}(c'),\mathsf{s}'_\tau,\bot\rangle,\sigma',\mathsf{fs}',\mathsf{md},\mathsf{Id}\end{array}}$$

$$\frac{\begin{array}{c}(\underline{\mathsf{THREAD{-}GTRANSACTION{-}COMMIT}})\\ \mathsf{md}'{=}\mathsf{md}[id{\mapsto}(\mathsf{beg},\mathsf{Now}(),\gamma_\mathsf{R},\gamma_\mathsf{W},\gamma_\mathsf{D},\mathcal{G})]\end{array}}{\begin{array}{c}\langle\tau,\epsilon,id{:}\mathsf{gablk}(\epsilon),\mathsf{s}_\tau,\bot\rangle,\sigma,\mathsf{fs},\mathsf{md},\mathsf{Id}\\ \xrightarrow{\mathsf{GCMT}}\\ \langle\tau,\epsilon,\epsilon,\mathsf{s}_\tau,\bot\rangle,\sigma,\mathsf{fs},\mathsf{md}',\mathsf{Id}\end{array}}$$

Figure 6.11: Guaranteed Transaction Command Rules.

- The read and write set of the guaranteed transaction are conservatively over approximated. $\mathsf{Reads} \stackrel{\mathsf{def}}{=} C \times \mathsf{Store} \times \mathsf{State} \to \mathsf{LocationSet}$ and $\mathsf{Writes} \stackrel{\mathsf{def}}{=}$

$C \times \mathsf{Store} \times \mathsf{State} \rightarrow \mathsf{LocationSet}$ return the transitive closure of memory locations the guaranteed transaction reads and respectively writes. Existing analyses such as [Jenista and Demsky, 2009] and [Ghiya and Hendren, 1996] can compute this information efficiently, although the intended use case of guaranteed transactions is on privatising relatively small object graphs.

- The predicate $\mathsf{GConflict} \stackrel{\mathsf{def}}{=} \mathsf{LocationSet} \times \mathsf{MD} \rightarrow \mathsf{Bool}$ is true if the write set of the guaranteed transaction conflicts with the dataset of an actively executing guaranteed transaction.

- The operations Reads, Writes and GConflict are executed under a single global lock atomicity semantics. Note that during the invocations of these respective functions they abort any conflicting active transaction.

- Beginning the guaranteed transaction makes use of the intermediate construct gblk which is associated a fresh unique identifier.

- The metadata mapping is updated to include the read and write set of the guaranteed transaction. The metadata is identified as modelling a guaranteed transaction via the use of the Coord value $\mathcal{G}$.

(THREAD−GTRANSACTION−BLOCK) is applied if the write set of the guaranteed transaction conflicts with the dataset of an actively running guaranteed transaction. The thread blocks until its guaranteed transaction can be run. Note that guaranteed transactions would take on a similar semantics to that presented in [Harris et al., 2005]. That is, they only try to execute again when an actively running guaranteed transaction which conflicts with the blocking guaranteed transaction completes execution.

(THREAD−GTRANSACTION−IN) executes a command under a guaranteed transaction semantics. The semantics are identical to (THREAD−LOCK−IN). The rule (THREAD−GTRANSACTION−COMMIT) commits a transaction which simply involves updating the completion time of the guaranteed transaction's metadata.

6.2.2 Parallel Composition

The parallel composition rule for transactions and guaranteed transactions is given in Figure 6.12. The rule is similar to Figure 4.17 so we only describe its differences.

- The thread configurations in I model guaranteed transactions which block. Those in J will execute due to their write sets not conflicting with actively executing guaranteed transactions.
- The transitions that the guaranteed transactions in J go through in the box labelled C are as follows:
  - A guaranteed transaction begins its execution, configuration T'_j, as it does not conflict with another active guaranteed transaction (see label D). This entails the update of the metadata and unique label components.
  - The constituent commands of the guaranteed transaction are executed resulting in T''_j which differs to T'_j in that its constituent commands may have allocated memory, updated the thread store and/or global state.

– The guaranteed transaction commits in T_j''' which sees the metadata component updated to reflect the guaranteed transaction's commit time, and the command which followed the guaranteed transaction being set as the thread's active command.

The intuition behind (PROGRAM−PARALLEL−COMPOSITON) is similar to the one given in Figure 4.17. That is, the active threads are partitioned according to the coordination semantics of their active commands. We assume that the threads in M and J are executing transactions and guaranteed transactions that do not conflict with the other transactions and respectively guaranteed transactions in their group of threads. The transactions in K either conflict with a transaction in M or a guaranteed transaction in J. The guaranteed transactions in I conflict with one or more actively executing guaranteed transactions in J.

6.3 Moverness

We redefine the definitions of moverness originally given in Chapter 5 but for transactions, guaranteed transactions and uncoordinated commands.

Definition 6.1 (Free Mover)**.** Let λ_1^+ be the sequence of actions issued by the command c_1 and λ_2^+ those issued by c_2, such that $c_1 \mid\mid c_2$. The constituent actions of λ_1^+ and λ_2^+ can freely move with respect to one another if and only if:

1. either c_1 or c_2 issue its sequence of actions under an uncoordinated semantics; or

2. c_1 and c_2 issue their respective sequence of actions via guaranteed transaction such that c_1 and c_2's guaranteed transactions do not conflict; or

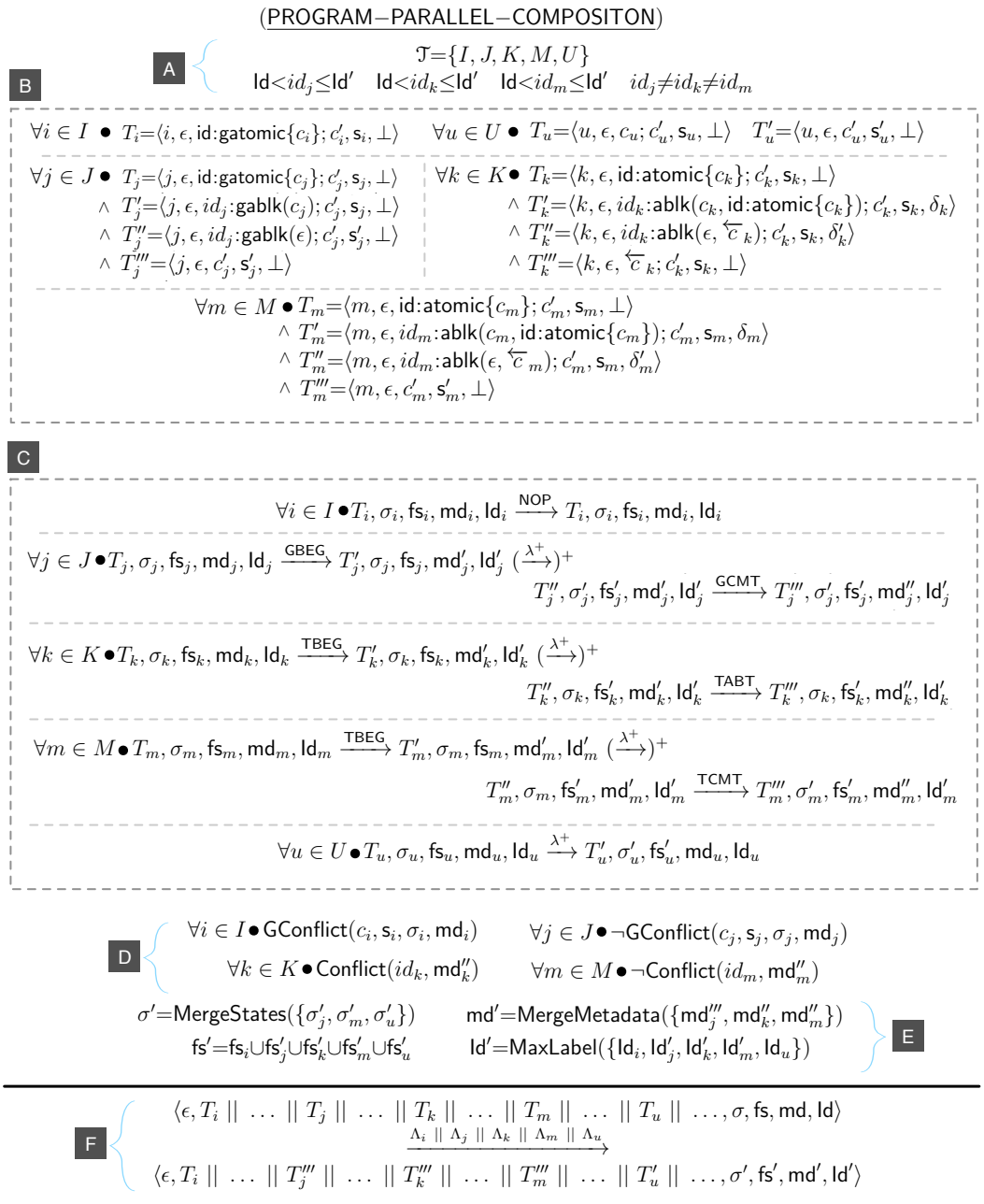


Figure 6.12: Parallel Composition Rule for Transactions and Guaranteed Transactions.

3. c_1 issues its sequence of actions under a guaranteed transaction semantics and c_2 under a transactional semantics, such that c_1's transaction does not conflict with c_2's guaranteed transaction; or

4. c_1 and c_2 issue their respective sequence of actions transactionally, such that c_1 and c_2's transactions do not conflict.

□

Definition 6.2 (Left Mover)**.** Let λ_1^+ be the sequence of actions issued by a command c_1 and λ_2^+ be those issued by c_2, such that $c_1 \mid\mid c_2$. Further, let λ_1^+ be issued under a transactional semantics and λ_2^+ under a guaranteed transaction semantics, such that there exists a write to a memory location ℓ in λ_2^+ and an access of ℓ in λ_1^+. We say that the sequence λ_2^+ moves to the left of λ_1^+, λ_2^+ λ_1^+, due to the weaker (abortable) semantics of transactions. That is, the constituent actions of λ_2^+ are guaranteed to take place before any of those in λ_1^+. □

Definition 6.3 (Right Mover)**.** A right mover is the mirror of a left mover. Let λ_1^+ be the sequence of actions issued by a command c_1 and λ_2^+ be those issued by c_2, such that $c_1 \mid\mid c_2$. Further, let λ_1^+ be issued under a transactional semantics and λ_2^+ under a guaranteed transaction semantics, such that there exists a write to a memory location ℓ in λ_2^+ and an access of ℓ in λ_1^+. We say that the sequence λ_1^+ moves to the right of λ_2^+, λ_2^+ λ_1^+, due to the weaker (abortable) semantics of transactions. That is, the constituent actions of λ_2^+ are guaranteed to take place before any of those in λ_1^+. □

Definition 6.4 (Both Mover)**.** Guaranteed transactions and transactions are both movers with respect to themselves. Let λ_1^+ be the sequence of actions issued by a command c_1 and λ_2^+ be those issued by c_2, such that $c_1 \mid\mid c_2$.

- if λ_1^+ and λ_2^+ are issued under a transactional semantics, and the accesses issued by λ_1^+ and λ_2^+ result in a conflict, then:
  - λ_1^+ can move to the left of λ_2^+, λ_1^+ λ_2^+ (c_1 commits, c_2 aborts); or
  - λ_1^+ can move to the right of λ_2^+, λ_2^+ λ_1^+ (c_2 commits, c_1 aborts).
- if λ_1^+ and λ_2^+ are issued under a guaranteed transaction semantics, and the accesses issued by λ_1^+ and λ_2^+ conflict, then:
  - λ_1^+ can move to the left of λ_2^+, λ_1^+ λ_2^+ (c_1 commits, c_2 blocks); or
  - λ_1^+ can move to the right of λ_2^+, λ_2^+ λ_1^+ (c_2 commits, c_1 blocks).

□

We do not treat moverness of locks and guaranteed transactions, despite such properties being trivial. For example, semantically speaking, both locks and guaranteed transactions are of equal strength. Therefore, should a guaranteed transaction and lock not conflict, that is the guarantee transaction not access the mutex used by the lock, then the actions of the respect lock and guaranteed transaction are free movers. By contrast, should the lock and transaction conflict, then they are both movers.

6.4 Applying Guaranteed Transactions

We now apply guaranteed transactions to the problem of applying an irreversible operation to a list suffix, a problem similar to that presented in Spear et al. [2007], to demonstrate their application and advantages, which we describe as we proceed during presentation of the example. The basic outline of the problem is as follows:

find a list suffix and privatise it, apply an operation to that suffix, then publicise that suffix. Guaranteed transactions greatly simplify the problem, and in conjunction with the moverness properties of guaranteed transactions (discussed in section 6.3 guarantee that reads issued via guaranteed transactions or transactions to the same memory observe the correct value. Importantly, as we have explained previously, guaranteed transactions handle privatisation/publication without having the programmer resort to explicit application of the idioms, and the semantics are only applied should they be required: for example, the suffix won't be privatised if the suffix's data is only read. Figure 6.13 gives the basic intuition of our example pictorially.

Achieving the semantics required for Figure 6.13 using the privatisation/publication idioms generally requires a pattern sketched out in Figure 6.14. Here, the first transaction finds the suffix, privatises it, the operation is then performed on its members non-transactionally, finally the second transaction publicises the previously provatised list suffix.

To give context to our problem we will work with a simple singly linked list data structure as shown in Figure 6.15. The data structure itself is trivial: nodes are added to the head of the list via `add`, in addition to supporting a more interesting method `serialise_suffix` which is an instance of the problem outlined in Figure 6.13. `serialise_suffix` attempts to write the members of the suffix specified by the user to disk (an irreversible operation) after it has mutated their values. The mutation is important as it will trigure a serialised semantics should mutlitple threads invoke `serialise_suffix` on the same `LinkedList` instance; if the mutation did not exist then the semantics of guaranteed transactions would permit calls to `serialise_suffix` to take place concurrently as their invocation

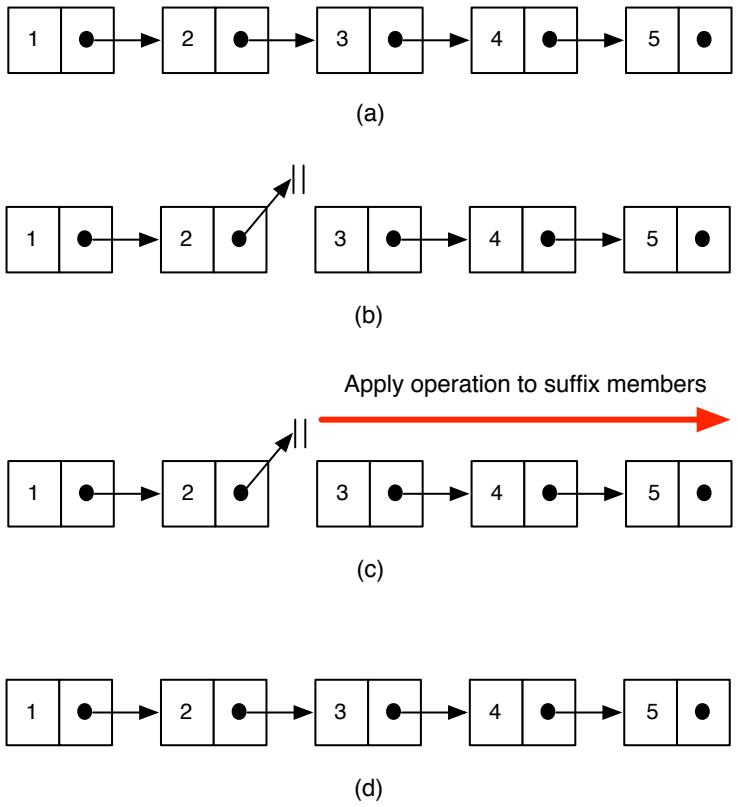


Figure 6.13: (a) Instance of a singly linked list; (b) privatise list suffix at 2; (c) apply an operation upon the suffix members; (d) publicise the list suffix.

```
atomic {
  // find list suffix, if possible
  // privatise it to the current thread
}

// apply operation to suffix members

atomic {
  // publicise the suffix
}
```

Figure 6.14: Pseudo steps for attaining the semantics required for Figure 6.13 using the privatisation and publication idioms.

does not intersect with the other's dataset.

Example 6.1 (Serialised guaranteed transactions)**.** Consider the following pro-

```
class Node {
  int value;
  Node head;
}

class LinkedList {
  Node head;

  void add(int value) {
     Node n := new Node;
     n.value := value;
     n.next := this.head;
     this.head := n;
  }


  void serialise_suffix(int value) {
    gatomic {
      Node n := this.head;
      while (n != null && n.value != value) {
        n := n.next;
      }
      if (n != null && n.value == value) {
        while (n != null) {
          n.value := n.value + 1;
          Disk.Write(n.value);
          n := n.next;
        }
      }
    }
  }
}
```

Figure 6.15: Singly linked list entailing a privatising/publicising operation on the members of a user-defined suffix. `serialise_suffix` mutates the members of a suffix in addition to applying an irreversible operation on those members via writing them to disk courtesy of of `Disk.Write`.

gram:

```
   LinkedList l; l := new LinkedList;

l.add(1); l.add(2); l.add(3); l.add(4);
```

Thread 1	**Thread 2**
`l.serialise_suffix(3);`	`l.serialise_suffix(2);`

Here, there are only two outcomes due to each guaranteed transaction writing to memory the other accesses: either thread 1's guaranteed transaction executes

first followed by thread 2's or vice versa. Consequently, the state of disk will be $4, 3, 2$ then $4, 3$ or $3, 2$ then $4, 4, 3$. Recall that (THREAD−GTRANSACTION−BEGIN) first checks its write set will not intersect with other currently executing guaranteed transactions, otherwise it blocks via (THREAD−GTRANSACTION−BLOCK). Therefore, for our first case (THREAD−GTRANSACTION−BEGIN) for thread 1's guaranteed transaction is applicable, but (THREAD−GTRANSACTION−BLOCK) is applied for thread 2's as its write set intersects with that of thread 1's actively executing guaranteed transaction. □

An important property of guaranteed transactions is there observation semantics which are defined via moverness and may be mapped to a memory model such as Java's in the same way as shown in Chapter 5. The side effect of this property is that transactions and guaranteed transactions are guaranteed to observe the writes of committing instances. For example, if in Figure 6.15 we were to ammend the definition of `add` to encapsulate its commands within a transaction, as shown in Figure 6.16, then a concurrent invocation of `serialise_suffix` on the same list instance would have its reads and writes be related to those of the transaction. That is, the transaction would observe the writes issued by the guaranteed transaction, as a guaranteed transaction is a left mover with respect to a transaction.

Example 6.2 (Observation semantics of transactions and guaranteed transactions)**.** Consider the following program:

```
   LinkedList l; l := new LinkedList;
l.add(1); l.add(2); l.add(3);
```

```
class LinkedList {
  Node head;

  void add(int value) {
     atomic {
       Node n := new Node;
       n.value := value;
       n.next := this.head;
       this.head := n;
     }
  }


  void serialise_suffix(int value) {
    gatomic {
      Node n := this.head;
      while (n != null && n.value != value) {
        n := n.next;
      }
      if (n != null && n.value == value) {
        while (n != null) {
          n.value := n.value + 1;
          Disk.Write(n.value);
          n := n.next;
        }
      }
    }
  }
}
```

Figure 6.16: Transactional addition of a value to an instance of `LinkedList`.

Thread 1	**Thread 2**
`l.add(4);`	`l.serialise_suffix(2);`

Here, thread 2's guaranteed transaction aborts thread 1's transactional operation due to transactions being right movers with respect to guaranteed transactions. Consequently, thread 1's transaction observes any writes made by thread 2's guaranteed transaction. □

6.5 Summary

In this chapter we have presented guaranteed transactions which are an alternative to the privatisation and publication idioms. Guaranteed transactions are not a replacement for all application instances of the privatisation/publication idioms but do provide a convenient and intuitive replacement when wishing to execute operations on data with simple object graphs. Guaranteed transactions can also replace locks in such scenarios. We demonstrated the application of guaranteed transactions by applying an irreversible operation to a linked list. In cases when the data which a guaranteed transaction operates upon has a complex object graph the system can revert to a single global lock atomicity semantics while preserving the simpler semantics that guaranteed transactions afford. Guaranteed transactions always give the user run once semantics while preserving object graph reachability invariants. Guaranteed transactions are a type of transaction so the programmer can define the semantics of his concurrent program using the simpler transactional programming model.

In this part of the thesis we have given three contributions: a low-level word-based small-step operational semantics for a programming language that supports locks and transactions, and transactions and guaranteed transactions; moverness definitions for locks, transactions and guaranteed transactions; and a safer means of a strong coordination semantics via the use of guaranteed transactions. We found the definition of a low-level semantics for transactions and locks to be a clear omission from the current literature, which we have tried to address in our work. We also found that mixing locks [Dijkstra, 1968] and transactions [Shavit and Touitou, 1995] results in a particularly complex but powerful semantics. To

simplify the semantics we used moverness [Barnett and Qin, 2012a] to generalise the observational properties of read actions which we found to be a particularly elegant solution. Guaranteed transactions [Barnett and Qin, 2012b] are an attempt to reduce the complexity of mixing transactions with a stronger coordination semantics without recourse to locks or the privatisation/publication idioms [Spear et al., 2007].

Part II

Static Reasoning

In this part of the thesis a program analysis is presented that guarantees the data-race-freedom (DRF) of fine-grained accesses in programs that use locks, transactions or both to coordinate accesses to memory. The presented framework entails two main steps. (i) *Static Execution*: a program is statically executed to determine the memory it allocates and the accesses (*access requirements*) it issues to that memory. The key artefact of a static execution is an *access mapping* which maps each memory location allocated by a program to its access requirements. (ii) *Isolation Algorithm*: isolation is checked for in the semantic information encapsulated by the access mapping from (i). Access isolation can be checked irrespective of whether accesses to the same memory use locks, transactions or both.

Chapter 7: A brief introduction to the problems of accessing shared memory using multiple coordination semantics is given, along with key definitions. The chapter concludes by showing a trivial application of our framework to a simple program.

Chapter 8: The syntax of the programming language that we use is described. Accesses to shared memory are issued under a lock, transactional or uncoordinated semantics.

Chapter 9. We describe how a program's stack and heap memory is represented. We then show how accesses to such memory are modelled by access requirements.

Chapter 10: The rules that drive a program's static execution are given. Application of each rule results in a set of access requirements being issued and stored in an incrementally built access mapping. We then present our

isolation algorithm which guarantees that all access requirements within the access mapping are isolated.

Chapter 7

Introduction

7.1 Isolation

Accesses (reads and writes) issued to the same memory in a concurrent program need to be *isolated* via the use of coordination, e.g. a lock [Dijkstra, 1983] or transaction [Shavit and Touitou, 1995]. Accesses are isolated if and only if their issuing coordination semantics prohibits them being scheduled concurrently. Failure to isolate accesses issued to the same memory introduces *data races* [Unger, 1995]. If accesses issued by distinct threads to a specific memory location are isolated then those accesses are data-race-free (DRF). If all accesses issued by a program are isolated then the program is DRF.

Definition 7.1 (Isolation of Concurrently Issued Accesses)**.** Two concurrently issued accesses a_1 and a_2, $a_1 \mid\mid a_2$, to a memory location ℓ are isolated if and only if, should they be scheduled concurrently, guarantee the total ordering $a_1\ a_2$ or $a_2\ a_1$. □

Intuitively, the accesses issued by a specific thread are isolated with respect

to all other accesses that thread issues. This is due to program order (Section 2.3.1).

Definition 7.2 (Program Order)**.** Taken in isolation the accesses issued by each thread form a total ordering known as program order. For example, let a_1 be a write and a_2 be a read of a memory location ℓ issued by a thread 1. Further, let the order in which a_1 and a_2 are issued by thread 1 be $a_1 \ldots a_2$, that is a_1 appears before a_2 in thread 1's program order. We assert that a_2 observes the value of ℓ written by a_1 unless a more recent intervening write of ℓ, a', exists in thread 1's program order such that $a_1 \ldots a' \ldots a_2$, in which case a_2 observes the value of ℓ written by a'. □

7.2 Isolation of Concurrently Issued Accesses

Locks and transactions provide the necessary semantics for isolating *most* shared memory accesses: locks (i) are suitable for executing irreversible and compute-bound operations; and (ii) offer an alternative when the overhead of transactionally accessing memory is too high. By contrast, transactions: (i) simplify component composition [Harris et al., 2005]; and (ii) alleviate the programmer from the error-prone maintenance of isolation invariants [Unger, 1995; Zöbel, 1983].

Reasoning about the isolation of concurrent programs that use locks *and* transactions to coordinate accesses to memory is particularly challenging. Here, the key issue is the granularity upon which isolation pivots: accesses issued by a lock are typically protected by a *mutex* (a binary semaphore [Dijkstra, 1968]); by contrast, a transaction entails multiple conceptual locks which are only acquired if another transaction accesses the same memory [Shavit and Touitou, 1995]. Stati-

cally reasoning about access isolation in programs that use locks and transactions to isolate accesses is extremely difficult, particularly in languages that offer weak immutability and sharing semantics, such as Java and C++.

To understand when accesses are isolated we will abstract the semantics given in Chapter 4 using the following definitions.

Definition 7.3 (Isolation of Lock and Transactional Accesses)**.** Two concurrently issued coordinated accesses a_1 and a_2 to a memory location ℓ, $a_1 \mid\mid a_2$, where either a_1 and/or a_2 is a write are isolated if and only if:

1. $a_1 = \mathsf{atomic}\{\ell\}$ and $a_2 = \mathsf{atomic}\{\ell\}$; or
2. $a_1 = \mathsf{sync}(\ell_1)\{\ell\}$ and $a_2 = \mathsf{sync}(\ell_2)\{\ell\}$, where $\ell_1 = \ell_2$; or
3. $a_1 = \mathsf{sync}(\ell_1)\{\ell\}$ and $a_2 = \mathsf{atomic}\{\ell; \ell_2\}$, where $\ell_1 = \ell_2$.

□

Definition 7.4 (Isolation of Concurrently Issued Uncoordinated Accesses)**.** Two concurrently issued uncoordinated accesses a_1 and a_2, $a_1 \mid\mid a_2$, to a memory location ℓ are never isolated. That is, the schedules $a_1 a_2$, $a_2 a_1$ and $a_1 \mid\mid a_2$ are all possible. □

Definition 7.5 (DRF of Concurrent Reads)**.** Two concurrent reads of a memory location ℓ by the accesses a_1 and a_2, $\mathsf{Any}(a_1) \mid\mid \mathsf{Any}(a_2)$, are trivially DRF. Where, $\mathsf{Any}(a)$ is used to denote that the access a can be issued under any semantics. This holds because neither thread mutates the value of ℓ. □

Definition 7.6 (DRF)**.** Two accesses a_1 and a_2, $a_1 \mid\mid a_2$, to a memory location ℓ are DRF if and only if:

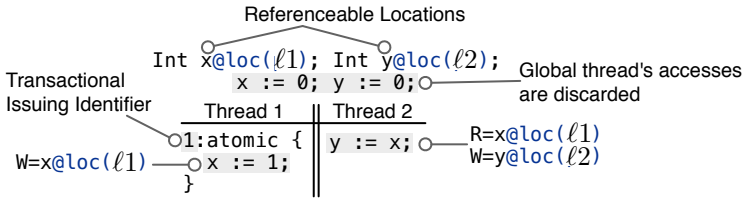


Figure 7.1: A simple program annotated with the inferred memory locations ($\ell 1$ and $\ell 2$) for the global variables x@loc($\ell 1$) and y@loc($\ell 2$). Execution of thread 1's assignment results in a write (W) of $\ell 1$; Executing thread 2's assignment results in a read (R) of $\ell 1$ and a write of $\ell 2$.

- $a_1 a_2$, that is a_1 and a_2 are related only by program order; or
- in $a_1 \mid\mid a_2$, $a_1 = \mathsf{Any}(:= \ell)$ and $a_2 = \mathsf{Any}(:= \ell)$; or
- in $a_1 \mid\mid a_2$, a_1 and a_2 are isolated via use of locks, transactions or both. □

The requirement for isolating accesses is only of importance when several threads access a memory location ℓ, and at least one of those threads writes ℓ.

7.3 Example

Figure 7.1 shows a simple program annotated with information inferred from its static execution. Each referenceable location (x and y) has an associated memory location: x@loc($\ell 1$) and respectively y@loc($\ell 2$), where $\ell 1 \neq \ell 2$ are memory locations and x@loc($\ell 1$) reads as "x resides at the memory location $\ell 1$." The goal of our analysis is to model the type of accesses issued to $\ell 1$ and $\ell 2$ during its static execution. The accesses issued by the program to $\ell 1$ and $\ell 2$ are modelled by *access requirements*. It is best to think of an access requirement as a closed

form of access which captures the issuing thread; a numerical value (*scale*) that distinguishes the type of access – a fraction between 0 and 1 for a read, and 1 for a write [Boyland, 2003]; the coordination type – transactional, lock-based or uncoordinated; and the identifier of the issuing coordination instance. The primary purpose of access requirements is to facilitate a uniform reasoning of access isolation irrespective of whether accesses to the same memory location were issued under an uncoordinated, lock or transactional semantics.

The access requirement that models the execution of thread 1's write of $\mathtt{x}@\mathsf{loc}(\ell 1)$ in Figure 7.1 is the quadruple $(\mathsf{TID}{=}1, \mathsf{Scale}{=}1, \mathsf{Coord}{=}\mathcal{A}, \mathsf{Issuer}{=}1)$, where: TID is the identifier of the issuing thread (Thread 1); Scale is the type of access (1, a write); Coord is the type of coordination the access is issued under ($\mathcal{A}$, a transaction); and Issuer the identifier of the issuing transactional instance (1). Locks and transactions have an Issuer value to facilitate isolation checks when an memory location is accessed by locks and transactions. Executing thread 2's assignment results in: (1) a read issued to $\mathtt{x}@\mathsf{loc}(\ell 1)$, $(\mathsf{TID}{=}2, \mathsf{Scale}{=}\epsilon, \mathsf{Coord}{=}\bot, \mathsf{Issuer}{=}\bot)$, where $0{<}\epsilon{<}1$ is a fraction that represents a read, and $\bot$ for Coord and Issuer denotes the read is issued under no coordination semantics; and (2) a write issued to $\mathtt{y}@\mathsf{loc}(\ell 2)$, $(\mathsf{TID}{=}2, \mathsf{Scale}{=}1, \mathsf{Coord}{=}\bot, \mathsf{Issuer}{=}\bot)$. The access mapping instance am that models the accesses issued by the program is:

$$[\ell 1 \mapsto \{(\underline{1}, 1, \mathcal{A}, 1), (\underline{2}, \epsilon, \bot, \bot)\}, \ell 2 \mapsto \{(2, 1, \bot, \bot)\}] \subseteq am$$

Where, the domain of am is the set of memory locations the program allocates

($\ell 1$ and $\ell 2$) and co-domain a set of access requirements on those memory locations. In this instance *am* is rejected by our isolation algorithm. The sum of scales (highlighted) on $\ell 1$ exceeds 1, $1+\epsilon>1$, as such we know at least one access to $\ell 1$ is a write. Closer inspection reveals that two threads (underlined, TIDs 1 and 2) access $\ell 1$, therefore all accesses to $\ell 1$ must be coordinated. We note thread 1's access to $\ell 1$ as being transactional and thread 2's as being uncoordinated. Consequently, the program is rejected as thread 1's transactional write of `x`@loc($\ell 1$) may be scheduled concurrently with thread 2's uncoordinated read of `x`@loc($\ell 1$), resulting in a data race [Unger, 1995].

7.4 Summary

Access isolation in a concurrent program is critical: failing to correctly isolate accesses to shared memory that is accessed by multiple threads, where at least one of those accesses is a write, may lead to a data race. A data race can have serious logical and security consequences in a program so should be prevented at all costs. Correctly isolating accesses in a program that uses just locks to isolate accesses has been shown in the past to be complex. Attaining access isolation in purely transactional programs is simpler as the programmer does not need to specify isolation invariants (e.g., mutexes, etc.) but the programmer must still issue accesses to shared memory transactionally. A programmer may wish to use both locks and transactions in the same program, applying each in situations which they are mutually appropriate: locks incur a low runtime cost and afford run once semantics; by contrast, transactions simplify component composition and in cases when performance is not the ultimate concern, provide a far simpler

isolation mechanism than locks. Unfortunately, reasoning about access isolation in a program that uses both locks and transactions is complex. For example, the programmer must reason not only about the isolation of accesses issued under the same coordination semantics but also those issued under distinct coordination semantics. We present a framework for automatically reasoning about the access isolation of such programs in this part of the thesis.

Chapter 8

Programming Model

The language that Part II of the thesis is based upon is a simplification of that used in Part I. The language presented here is driven by what is feasibly computable for determining the DRF of a program that uses locks, transactions or both to coordinate accesses to shared memory in a system that supports objects, method calls and unrestricted mutation of memory.

8.1 Programming Language

8.1.1 Core Language.

Locks [Dijkstra, 1983] and transactions [Shavit and Touitou, 1995] (see Section 2.2) are used to coordinate accesses to memory. A lock is described by $\mathsf{sync}(v)\{c\}$ where v is a variable that acts as a mutex and c the program text which it protects. Transactionally executing a command c is performed by $\mathsf{atomic}\{c\}$. Transactions are weakly isolated, out-of-place and conflict detection is at the granularity of memory locations. The isolation of accesses issued by nested locks and mutually

Core Language

Program	::=	Class-Decl* (Type v)$^+$ (v := new cn | $v := i_l$ | $v.f := i_l$)$^+$ ($v.m(i_l?)$@nodefer)* (C || ... || C)
Class-Decl	::=	Class-Ann class cn { (Type v)$^+$ Meth-Decl* }
Type	::=	cn | Int
Meth-Decl	::=	m((Type a)?) { (Type v)* C $_m$ }
$b \in$ BExpr	::=	$v \neq$ null | $v =$ null
$c \in$ C	::=	$v := x$ | $v := i_l$ | id:atomic{c} | id:sync(v){c} | $v.m(i_l?)$ | c;c'
$c_m \in$ C $_m$	::=	v := new cn | $v.f := x$ | $v.f := x.f$ | $v := x.f$ | Loop-Space-Ann while b {c_m} | print($v.f$) | c_m; $c_{m'}$

Memory Annotations

Class-Ann	::=	@object−space[fields=f^+ (;dynamic=fn)?] (@serialise[$m_1 < \cdots < m_n$])?
Loop-Space-Ann	::=	@iter−space[fn]
Mem-Fn	::=	$locs$ fn (E, val) { Mem-Pred }
Mem-Pred	::=	null | ℓ

Figure 8.1: Abstract Syntax of the Core Programming Language and Memory Annotations.

nested locks and transactions cannot be checked, we discuss why in 10. Nested transactions are flattened as in Part I. The metavariables v and x range over variables, i_l over integer literals (variables of type Int), cn over user defined classes, m over methods and $v.f$ over accesses to the field f defined by the receiver v's type. *, $^+$ and ? denote zero-or-more, one-or-more and respectively zero-or-one occurrences. A program's structure Program entails a sequence of class and global variable declarations, their initialisation and a parallel composition of threads. Classes Class-Decl are permitted to facilitate the checking of more advanced programs, as shown in Appendix B. Class methods are used to mutate values of memory which hold references to other objects. This restriction permits a simple

reasoning of when writes are required to be observed, which is particularly important for data structures like linked lists. The underlined parts of the syntax are a side-effect of our program text preprocessing. Note that the unique label id associated with a lock or transaction is statically bound, by contrast to Part I where id was a label that was bound dynamically to a unique identifier.

8.1.2 Memory Annotations.

A class is decorated with Class-Ann which comprises two parts. (1) @object−space describes the memory space that an object of its decorating type will occupy: the memory location associated with each of its fields, fields, in addition to any memory the class dynamically allocates, dynamic. (2, optionally) @serialise which describes a total order over a class's member methods. A *memory function* Mem-Fn fn computes the dynamic memory space of an object. It is defined as a sequence of structural predicates over the value val (the literal value null or a memory location ℓ) and returns a set of memory locations $locs$. We use fn on its own to be a metavariable over memory function application. A while loop is decorated with Loop-Space-Ann which specifies the dynamic memory the while loop reads. We give a thorough treatment of memory annotations is given in Chapter 9.

8.1.3 Preprocessing.

Lock and transactional instances are given a unique identifier id, id:atomic$\{c\}$ and respectively id:sync$(v)\{c\}$. Method invocations by the main thread are annotated with @nodefer. Methods annotated with @nodefer are executed immediately upon

being encountered within the program text.

8.2 Summary

The language presented in this chapter allows the programmer to create sufficiently complex programs that make use of dynamically allocated data structures, e.g. linked lists. The key focus of the language is on mutation and the use of locks [Dijkstra, 1968] and transactions [Shavit and Touitou, 1995], rather than a comprehensive feature list. Mutation helps to form interesting object graphs which are inherently shared between several threads. Locks and transactions are used to coordinate accesses to the memory which the object graph occupies. The data-race-freedom of these accesses is the subject of the static analysis we present in subsequent chapters. Memory annotations, which we cover in Chapter 9, augment the core programming language and are used to drive the static execution of a program.

Chapter 9

Memory and Memory Accesses

We now describe how the memory consumed by a program's stack and heap data is modelled, and how accesses to such data are captured by *access requirements*.

9.1 Memory

9.1.1 Stack Variables

A stack variable is associated with a memory location and the value null upon declaration. For example, the variable declaration `X v`, where `X` is the type of variable `v`, sees `v` associated with a pair whose first component is the fresh memory location ℓ, the stack slot address of `v`, and second component null. The term "fresh ℓ" denotes the memory location ℓ is unbound in a program's *free store*: the set of memory locations currently in use by a program. We use the mapping $\mathsf{Var} \stackrel{\mathsf{def}}{=} \mathsf{Variable} \rightarrow \mathsf{Location} \times \mathsf{Location}$ to map a variable to its stack location and value pair. Recall that null, along with all possible memory locations ℓ are valid instances of $\mathsf{Location}$.

Example 9.1 (Variable Declaration)**.** Let var be a variable mapping Var. Executing the variable declaration `X v` results in $[v \mapsto (\ell, \mathsf{null})] \subseteq var$, where ℓ is a fresh memory location. □

9.1.2 Heap Objects

The mapping $\mathsf{Object} \stackrel{\mathsf{def}}{=} \mathsf{Field} \to \mathsf{Location} \times \mathsf{Location}$ models the memory space of an allocated instance of a class. Each object is a mapping from a field name to a pair whose first component is the memory location of the field and second component its value. Each field specified by a class's @object−space.fields annotation resides at a distinct memory location within an object of that class. For example, allocation of a `Point` as given in Figure 9.1 results in `x` and `y` occupying distinct memory locations. The fields property of a class's @object−space annotation declares the immediate memory space of an object of its type and can be read as "the memory space occupied by allocating a `Point` is a memory location for `x` and a memory location for `y`." Because `Point` comprises data of literal types – integers – the fields property for @object−space is all that is required as the object graph of a `Point` object is fixed upon allocation. That is, the `x` and `y` fields of a `Point` object are leaf nodes in a program's object graph.

```
@object-space[fields=x,y]
class Point {
  Int x;
  Int y;
}
```

Figure 9.1: A simple `Point` class with fields for `x` and `y` coordinates.

Example 9.2 (Object Mapping)**.** Given the definition of `Point` in Figure 9.1, the

object mapping created as a result of the command `new Point` is $[\mathtt{x}{\mapsto}(\ell 1, \mathsf{null}), \mathtt{y}{\mapsto}(\ell 2, \mathsf{null})]$, where $\ell 1 \neq \ell 2$ and the initial value of each field of the object is null. The *memory space* of this object is $\{\ell 1, \ell 2\}$. $\square$

The memory location of the first field in the domain of an object, its *base location*, ℓ_{base}, is the start address of an object. This semantics is modelled on "plain old data" types in C/C++. That is, we treat an object like a basic `struct`. The mapping $\mathsf{Obj} \stackrel{\mathsf{def}}{=} \mathsf{Location} \rightarrow \mathsf{Object}$ maps the base address of an object to the object it refers to.

Example 9.3 (Object Base Location)**.** Let $[\mathtt{x}{\mapsto}(\ell 1, \mathsf{null}), \mathtt{y}{\mapsto}(\ell 2, \mathsf{null})]{\subseteq}pt$ be a `Point` object. The base location of pt is $\mathsf{fst}(pt(\mathsf{Head}(\mathsf{Dom}(pt)))) = \ell 1$. Where, $\mathsf{Head}(\{a, \dots\}){=}a$. $\square$

Example 9.4 (Allocation)**.** Let var be a variables mapping Var and obj an empty object mapping Obj such that $[v{\mapsto}(\ell 1, \mathsf{null})]{\subseteq}var$. Executing the command `v := new Point` results in $var'{=}var[v{\mapsto}(\ell 1, \ell 2)]$ and $obj'{=}obj[\ell 2{\mapsto}[\mathtt{x}{\mapsto}(\ell 2, \mathsf{null}), \mathtt{y}{\mapsto}(\ell 3, \mathsf{null})]]$. $\square$

The Var and Obj mappings are used to compute the memory space of a command in our static execution rules given Section 10.1.

Example 9.5 (Var and Obj for Computing Memory Locations Accessed)**.** Let var be a variables mapping Var and obj an object mapping Obj such that:

$$[v{\mapsto}(\ell 1, \ell 3), x{\mapsto}(\ell 2, \mathsf{null})]{\subseteq}var \quad [\ell 3{\mapsto}[\mathtt{x}{\mapsto}(\ell 3, \mathsf{null}), \mathtt{y}{\mapsto}(\ell 4, \mathsf{null})]]{\subseteq}obj$$

The memory locations accessed in the command `x := v.y` are $\mathsf{fst}(var(x))$, $\mathsf{fst}(var(v))$ and $\mathsf{fst}(obj(\mathsf{snd}(var(v)))(y))$. That is, the locations $\ell 2$, $\ell 1$ and $\ell 4$.

Where, $\mathsf{fst}((a,b))=a$ and $\mathsf{snd}((a,b))=b$. □

We present a syntactically more elegant way to access information such as a variable's memory location and value, etc. in Chapter 10. For now, use of the more verbose syntax gives a better understanding of how memory information is attained.

```
@object-space[fields=next,value]
class Node {
  Node next;
  Int value;
}

@object-space[fields=head;dynamic=nodes(E,head)]
@serialise[add < traverse]
class LinkedList {
  Node head;
  add(Int val) {
    Node n;
    n := new Node;
    n.value := val;
    n.next := this.head;
    this.head := n;
  }

  traverse() {
    Node curr;
    curr := this.head;
    @iter-space[object-space.dynamic]
    while (curr ≠ null) {
      print(curr.value);
      curr := curr.next;
    }
  }
}
```

Figure 9.2: An advanced application of our system. `Node` and `LinkedList` classes make use of @object−space, @serialise and @iter−space annotations.

A method of a class may allocate data, e.g. `add` in `LinkedList` given in Figure 9.2. Here, the fields property of the @object−space annotation alone is insufficient: the memory space a `LinkedList` object occupies is that of its member fields *and* that of the `Node` objects it allocates. A class that allocates heap data as a side-

effect of invoking one of its member operations must specify a *memory function* (Mem-Fn , Figure 8.1) via the dynamic property of the class's @object−space annotation. A memory function takes an environment E (described in Section 10.1) and location as arguments and returns the set of memory locations reachable from that value. Note that the only thing we need to be aware of for E at this moment in time is that it comprises an object mapping Obj. The memory function of `LinkedList` in Figure 9.2 is $\mathsf{nodes} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Location} \to \mathsf{LocationSet}$:

$$\mathsf{nodes}(\mathsf{E}, val) \stackrel{\mathsf{def}}{=} \begin{cases} \{\} & \textbf{if } val = \mathsf{null} \\ \{\ell_1, \ell_2\} \cup \mathsf{nodes}(\mathsf{E}, val_{\texttt{next}}) & \textbf{if } val \neq \mathsf{null} \wedge^{\dagger} \end{cases}$$

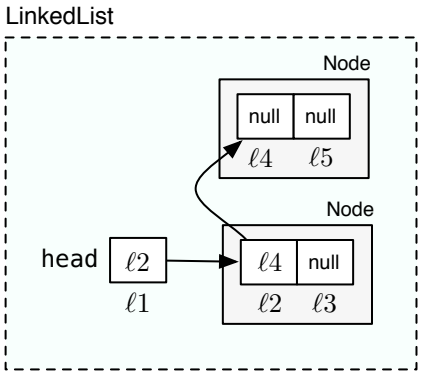
$$^{\dagger}[\ell_1 \mapsto [\mathsf{next} \mapsto (\ell_1, val_{\texttt{next}}), \mathsf{value} \mapsto (\ell_2, \mathsf{null})]] \subseteq \mathsf{E.Obj}$$

Where, the subscripted ℓs ℓ_1, ℓ_2 and ℓ_3 are metavariables over actual memory locations.

Example 9.6 (Computing the Dynamic Memory Space of a Linked List)**.** Given an instance *env* of E:

$$[\ell 1 \mapsto [\texttt{head} \mapsto (\ell 1, \ell 2)],$$
$$\ell 2 \mapsto [\texttt{next} \mapsto (\ell 2, \ell 4), \texttt{value} \mapsto (\ell 3, \mathsf{null})],$$
$$\ell 4 \mapsto [\texttt{next} \mapsto (\ell 4, \mathsf{null}), \texttt{value} \mapsto (\ell 5, \mathsf{null})]] \subseteq env.\mathsf{Obj}$$

Which models the following linked list:



We can compute its dynamic memory space by applying nodes with the value of `head`, $\ell 2$:

$$\begin{aligned} \mathsf{nodes}(env, \ell 2) = \quad & \{\ell 2, \ell 3\} \cup \quad \mathsf{nodes}(env, \ell 4) \\ & \{\ell 4, \ell 5\} \cup \qquad \mathsf{nodes}(env, \mathsf{null}) \\ & \{\} \end{aligned}$$

Which results in the set of memory locations $\{\ell 2, \ell 3, \ell 4, \ell 5\}$. □

9.1.3 Iteration Space

A loop such as the `while` construct is often used to iterate over a dynamic memory space, e.g. `traverse` in `LinkedList`. A loop must be decorated with an @iter−space annotation if it reads dynamic memory. For example, the `traverse` method in Figure 9.2 uses the memory function defined by `LinkedList`. Here, we are stating that `traverse`'s while loop reads all the dynamic memory allocated by a `LinkedList` object.

9.2 Memory Accesses

We now give a quick refresher of permissions [Boyland, 2003] which were briefly discussed in Section 2.2.1, and our enriched version of permissions which we call *access requirements*. Permissions are used to partition reads and writes. Access requirements extend permissions to encode the issuing coordination semantics of accesses.

9.2.1 Permissions

Permissions [Boyland, 2003] are used to partition reads from writes: a read requires *part* of a permission; by contrast, a write requires the *whole* of a permission.

$$\mathsf{Permission} \stackrel{\mathsf{def}}{=} \mathsf{Scale}\ \ell$$

Where $\mathit{Scale} \stackrel{\mathsf{def}}{=} \epsilon \mid 1$ and ℓ is a memory location. Using this formalism we can define reads and writes as follows:

$$\mathsf{Read} \stackrel{\mathsf{def}}{=} \epsilon\ \ell \quad \mathsf{Write} \stackrel{\mathsf{def}}{=} 1\ \ell$$

Where $0 < \epsilon < 1$. The sum of read scales ϵ on the same memory location forms a whole.

Example 9.7 (Applying Permissions)**.** Let us assume that `v` resides at memory location $\ell 1$ and `x` at memory location $\ell 2$.

```
v := 0; x := 0;
```

Thread 1	Thread 2
`x := v;`	`v := x;`

The permissions that model thread 1's accesses are: $1\ell2$ (write of `x`) and $\epsilon\ell1$ (read of `v`). The permissions that model thread 2's accesses are: $1\ell1$ (write of `v`) and $\epsilon\ell2$ (read of `x`). □

9.2.2 Access Requirements

An access requirement enriches a permission with additional access metadata: $\mathsf{AR} \stackrel{\mathsf{def}}{=} (\mathsf{TID}, \mathsf{Scale}, \mathsf{Coord}, \mathsf{Issuer})$, where $\mathsf{TID} \stackrel{\mathsf{def}}{=} \mathsf{Int}$ is a unique thread identifier, Scale is as defined previously, $\mathsf{Coord} \stackrel{\mathsf{def}}{=} \bot \mid \mathcal{A} \mid \mathcal{L}(\ell)$ is the coordination type and $\mathsf{Issuer} \stackrel{\mathsf{def}}{=} \mathsf{Int}$ the unique identifier id associated with a lock or transaction instance. The values of Coord are as follows: $\bot$ is uncoordinated; $\mathcal{A}$ is transactional; and $\mathcal{L}(\ell)$ is lock-based. The value $\mathcal{L}(\ell)$ *lock-contextualises* ℓ which is the memory location associated with the variable the lock is protected on. The last two components of an access requirement are $\bot$ when the access being modelled is uncoordinated.

Example 9.8 (Applying Access Requirements)**.** First, consider the same program from Example 9.7:

```
v := 0; x := 0;
```

Thread 1	Thread 2
`x := v;`	`v := x;`

Where `v` and `x` reside at the memory locations $\ell1$ and respectively $\ell2$. Let us now define an access mapping, $\mathsf{AM} \stackrel{\mathsf{def}}{=} \mathsf{Location} \to \mathsf{ARSet}$, to be a mapping from a memory location to a set of access requirements on that memory location.

Assuming am is an instance of AM we can model the accesses issued by previous program as:

$$[\ell 1 \mapsto \{(1, \epsilon, \bot, \bot), (2, 1, \bot, \bot)\}, \ell 2 \mapsto \{(1, 1, \bot, \bot), (2, \epsilon, \bot, \bot)\}] \subseteq am$$

am for this example reads as follows:

- An uncoordinated read by thread identifier 1 and uncoordinated write by thread identifier 2 is issued to $\ell 1$; and
- An uncoordinated write by thread identifier 1 and uncoordinated read by thread identifier 2 is issued to $\ell 2$. □

In Example 9.7 we may observe that a write and read are issued to both `v` and `x`. However, what we cannot determine is whether these writes and reads were issued by distinct threads or not. Assuming we can determine such information we are now tasked with determining whether or not the accesses to `v` and `x` are isolated. Permissions alone are insufficient for this task and rely heavily on external components such as type rules to (try) and determine such a property. As we will show later, reasoning about the isolation of a program that permits the use of several coordination semantics is complex. Furthermore, deferring this reasoning to a type system is challenging and in many cases not possible. Our response is to take a hybrid approach: access requirements capture the key data required to reason about access isolation and the task of the static rules is to build an access mapping. Reasoning about the isolation of a program is then handled by an isolation algorithm which inspects an access mapping.

We will now demonstrate the use of access requirements by informally reasoning about the isolation of a program, aided only by the access requirements which model the accesses it issues.

Example 9.9 (Understanding Access Requirements)**.** We informally reason about the AM am given in Example 9.8, which was as follows:

$$[\ell 1 \mapsto \{(1, \epsilon, \bot, \bot), (2, 1, \bot, \bot)\}, \ell 2 \mapsto \{(1, 1, \bot, \bot), (2, \epsilon, \bot, \bot)\}] \subseteq am$$

- $\ell 1$. The first question we may pose is "are any writes issued to $\ell 1$?" Clearly, we can see that one thread writes $\ell 1$, namely the thread with $\mathsf{TID} = 2$. Due to the previous answer we may subsequently ask "Does a single thread access $\ell 1$?" We observe that threads with TIDs 1 and 2 access $\ell 1$. It follows from our previous enquiries that two threads access $\ell 1$, with one of those accesses being a write. Consequently, we require the accesses to $\ell 1$ be isolated. Inspecting the accesses of both threads to $\ell 1$ we see that each is uncoordinated. Therefore, accesses to $\ell 1$ are not isolated as each thread's respective access of $\ell 1$ may be issued concurrently with respect to the other thread's access of $\ell 1$.
- $\ell 2$. Accesses to $\ell 2$ are not isolated due to a similar argument as $\ell 1$.

The key point of this example is that each question (and ones we have yet to pose) can be answered by just looking at the access requirements on a memory location. □

Example 9.9 gave a basic intuition of the role that access requirements play in

our framework. We have found that an access requirement captures just enough information to answer queries of access isolation in both simple and complex situations. In Section 10.2 we present an algorithm that mechanically reasons about the isolation of accesses issued to each memory location allocated by a program.

9.3 Summary

In this chapter we presented how our static analysis framework models the memory allocated by a program and how accesses issued to this memory are captured. Each variable and object field is associated with a unique memory location upon declaration/allocation. Objects have the same semantics as structs in C. Accesses issued by the program are captured by access requirements which are an extension of fractional permissions [Boyland, 2003]. We use fractional permissions as it offers an intutive means to partition reads from writes. Furthermore, fractional permissions make our isolation algorithm vastly simpler to construct as we can make use of basic arithmetic on permission scales. Each access requirement comprises the thread identifier that issued the access, the scale of the access (read or write), the coordination semantics the access was issued under (lock, transaction or uncoordinated) and the identifier of the coordination instance (if issued under a lock or tranasction) the access originated from. Access requirements encapsulate the necessary information required to make unambigous decisions about the data-race-freedom of accesses issued to the same memory, irrespective of the coordination semantics the accesses were issued under.

Chapter 10

Static Execution Rules and Isolation Algorithm

In this chapter we present static execution rules which compute the accesses issued to memory by each command. The result of their application is an access mapping whose domain is the set of memory locations allocated by the program, and co-domain the set of access requirements on those memory locations. The access mapping resulting from the static execution of a program is validated by our isolation algorithm (Section 10.2).

Note that, like in Part I, we give mainly informal discussions of the functions referenced throughout. See Appendix A for their formal definitions.

10.1 Static Execution Rules

Application of each rule executes a command. Execution of a command focuses specifically on its memory semantics. That is, the memory a command may

allocate and the memory it may access. The accesses a command issues are captured as access requirements in the program's incrementally constructed access mapping (See Chapter 9.

Example 10.1 (Rule Application)**.** Let us assume that x and y reside at the memory locations $\ell 1$ and respectively $\ell 2$. Static execution of the uncoordinated assignment $x := y$ by a thread with identifier 1, results in a read on $\ell 2$ and a write on $\ell 1$. These access semantics are encoded by the rules in an instance of an access mapping AM. Let am by an instance of AM that is used to execute the program which entails the command `x := y`. Execution of $x := y$ leaves am in the following state: $[\ell 1 \mapsto \{(1, 1, \bot, \bot)\}, \ell 2 \mapsto \{(1, \epsilon, \bot, \bot)\}] \subseteq am$. □

10.1.1 Environment

A command is executed in an environment $\mathsf{E} \stackrel{\mathsf{def}}{=} \mathsf{TID}; \mathsf{FS}; \mathsf{Issuer}; \mathsf{Coord}; \mathsf{Var}; \mathsf{Obj}; \mathsf{AM}; \mathsf{Dfr}$.

- $\mathsf{TID} \stackrel{\mathsf{def}}{=} \mathsf{Int}$ is the active thread identifier.
- $\mathsf{FS} \stackrel{\mathsf{def}}{=} \mathsf{LocationSet}$ is the free store of the program.
- $\mathsf{Issuer} \stackrel{\mathsf{def}}{=} \mathsf{Int}$ is the unique label id associated with each lock and transactional instance. For example, this would be 1 in $1{:}\mathsf{atomic}\{c\}$.
- $\mathsf{Coord} \stackrel{\mathsf{def}}{=} \mathcal{L} \mid \mathcal{A} \mid \bot$ is active coordination semantics. $\mathcal{L}$ indicates the active coordination semantics is a lock, $\mathcal{A}$ a transaction and $\bot$ signals that no coordination semantics are active. $\mathcal{L}$ is parameterised on a memory location ℓ, $\mathcal{L}(\ell)$, which denotes the memory location of the variable being used as the mutex the active lock is protected on.

- $\mathsf{Var} \stackrel{\mathsf{def}}{=} \mathsf{Variable} \rightarrow \mathsf{Location} \times \mathsf{Location}$ maps a variable to a pair whose first component is the memory location the variable resides and second component the variable's value.

- $\mathsf{Obj} \stackrel{\mathsf{def}}{=} \mathsf{Location} \rightarrow \mathsf{Object}$ maps the base memory location of an object to the object which it refers to. $\mathsf{Object} \stackrel{\mathsf{def}}{=} \mathsf{Field} \rightarrow \mathsf{Location} \times \mathsf{Location}$ maps a field to a pair whose first component is the memory location the field resides at and second component the field's value.

- $\mathsf{AM} \stackrel{\mathsf{def}}{=} \mathsf{Location} \rightarrow \mathsf{ARSet}$ is a mapping from a memory location to a set of access requirements issued to that memory location. See Chapter 9.

- $\mathsf{Dfr} \stackrel{\mathsf{def}}{=} \mathsf{DeferredMethodCallList}$, where DeferredMethodCallList is a list of DeferredMethodCall which contains all instances of the form $v.m(i_l?)$@ctxt. That is, Dfr is a list of deferred method calls. We explain this concept throughout the coming chapter.

Some of these components we have seen in Part I, such as TID, FS, Var, Obj, Object and Coord. We point out that the value $\mathcal{L}$ of Coord is only parameterised on a memory location, by contrast to Part I where $\mathcal{L}$ was parameterised on a memory location and handle count.

10.1.2 Notation

The expression E[Component=*value*] yields an environment that is the same as E but with Component bound to *value*. Each Component of the environment is referred to by the same name as its defining type. E.Component returns the value of Component in E. Component in rule premises is short for E.Component.

$\mathsf{fresh}\ \ell$ asserts $p \notin \mathsf{FS}$. A primed value, e.g. $value'$, indicates an updated version of $value$. Functions that require access to an environment component take the environment E as their first argument. A subscripted ℓ, e.g. ℓ_1, is a metavariable over memory locations. A non-subscripted ℓ, e.g. ℓ and $\ell 1$, denote actual memory locations. For example, ℓ_1 and ℓ_2 may both resolve to $\ell 1$, but $\ell 1$ and $\ell 2$ denote distinct memory locations. This is consistent with the presentation used in Part I.

10.1.3 Judgements

Judgements are of the form $\mathsf{E} \vdash c \Rightarrow \mathsf{E}'$. Where, E' is the environment yielded by executing c from an environment E. If a command c cannot be satisfied by the environment E from which it is to be executed then we have $\mathsf{E} \vdash c \Rightarrow \bot$. That is, the environment yielded from executing c is undefined. A command whose execution results in an undefined environment is conservatively labelled as not isolated.

10.1.4 Constructing Access Requirements

Most rules we present in this chapter add access requirements to $\mathsf{E.AM}$, so we define $\mathsf{Add_{AR}} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Scale} \times \mathsf{LocationSet} \to \mathsf{AM}$ which adds a new access requirement to each of the memory locations specified with the given scale.

Example 10.2 (Constructing Access Requirements for a Command)**.** Let env be an instance of E and $env.\mathsf{AM}$ be an empty mapping. Further, let the access requirements we wish to model be that issued by the command `x := y`, where `x` resides at memory location $\ell 1$ and `y` $\ell 2$. We assume the command

is executed under no coordination semantics by a thread with the identifier 1. We can construct these access requirements via am_R=$\mathsf{Add}_{\mathsf{AR}}(env, \epsilon, \{\ell 2\})$ and am_W=$\mathsf{Add}_{\mathsf{AR}}(env, 1, \{\ell 1\})$, where am_R and am_W differ with $env.\mathsf{AM}$ in that they contain the read, am_R, and respectively write, am_W, access requirements:

$$[\ell 2 \mapsto \{(1, \epsilon, \bot, \bot)\}] \subseteq am_R \quad [\ell 1 \mapsto \{(1, 1, \bot, \bot)\}] \subseteq am_W$$

An obvious problem is that the access mappings am_R and am_W each contain the access requirements the command issued. The access mapping that becomes the new value of $env.\mathsf{AM}$ is that of merging am_R and am_W. We do this via the function $\mathsf{MergeAMs} \stackrel{\mathsf{def}}{=} \mathsf{AM} \times \mathsf{AM} \to \mathsf{AM}$ which takes two access mappings whose domain and co-domain are to be merged and returns the result of their merging. Let am'=$\mathsf{MergeAMs}(am_R, am_W)$, where

$$[\ell 1 \mapsto \{(1, 1, \bot, \bot)\}, \ell 2 \mapsto \{(1, \epsilon, \bot, \bot)\}] \subseteq am'$$

□

10.1.5 Rules

We now present the static execution rules which are given in Figures 10.1, 10.2, 10.3 and 10.4, then describe their operation. Please see Appendix A for the definitions of all the functions referenced by the rules.

$$\frac{\text{fresh } \ell \quad \mathsf{FS}'{=}\mathsf{FS} \cup \{\ell\} \quad \mathsf{Var}'{=}\mathsf{Var}[v{\mapsto}(\ell, \mathsf{null})]}{\mathsf{E} \vdash \boxed{\text{Type } v} \Rightarrow \mathsf{E}[\mathsf{FS}{=}\mathsf{FS}'; \mathsf{Var}{=}\mathsf{Var}']} \quad (\text{VAR-DECL})$$

$$\frac{[v{\mapsto}(\ell_1, \mathsf{null}), x{\mapsto}(\ell_2, \mathsf{null})]{\subseteq}\mathsf{Var} \quad \{\ell_1, \ell_2\} \subseteq \mathsf{FS} \quad \mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E}, \epsilon, \{\ell_2\}) \quad \mathsf{AM}_W{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E}, 1, \{\ell_1\}) \quad \mathsf{AM}'{=}\mathsf{MergeAMs}(\mathsf{AM}_R, \mathsf{AM}_W)}{\mathsf{E} \vdash \boxed{v := x} \Rightarrow \mathsf{E}[\mathsf{AM}{=}\mathsf{AM}']} \quad (\text{ASSIGN-VAR-LITERAL})$$

$$\frac{[v{\mapsto}(\ell_1, \mathsf{null})]{\subseteq}\mathsf{Var} \quad \ell_1 \in \mathsf{FS} \quad \mathsf{AM}'{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E}, 1, \{\ell_1\}) \quad (obj, locs){=}\mathsf{CreateObject}(\mathsf{E}, cn) \quad \mathsf{FS}'{=}\mathsf{FS} \cup locs \quad \ell_{\text{base}}{=}\mathsf{Head}(locs) \quad \mathsf{Var}'{=}\mathsf{Var}[v{\mapsto}(\ell_1, \ell_{\text{base}})] \quad \mathsf{Obj}'{=}\mathsf{Obj}[\ell_{\text{base}}{\mapsto}obj]}{\mathsf{E} \vdash \boxed{v := \text{ new } cn} \Rightarrow \mathsf{E}[\mathsf{FS}{=}\mathsf{FS}'; \mathsf{Var}{=}\mathsf{Var}'; \mathsf{Obj}{=}\mathsf{Obj}'; \mathsf{AM}{=}\mathsf{AM}']} \quad (\text{NEW})$$

$$\frac{\mathsf{Dfr}'{=}(v.m(i_l?)@\mathsf{ctxt}[\mathsf{TID}{=}\mathsf{E.TID}; \mathsf{Coord}{=}\mathsf{E.Coord}; \mathsf{Issuer}{=}\mathsf{E.Issuer}]) :: \mathsf{Dfr}}{\mathsf{E} \vdash \boxed{v.m(i_l?)} \Rightarrow \mathsf{E}[\mathsf{Dfr}{=}\mathsf{Dfr}']} \quad (\text{METHOD-CALL-DEFER})$$

$$\frac{[v{\mapsto}(\ell_1, \ell_2)]{\subseteq}\mathsf{Var} \ \{\ell_1, \ell_2\}{\subseteq}\mathsf{FS} \quad \ell_1 \neq \ell_2 \quad \ell_2 \in \mathsf{Dom}(\mathsf{Obj}) \quad \mathsf{E}'{=}\mathsf{E}[\mathsf{TID}{=}@\mathsf{ctxt.TID}; \mathsf{Coord}{=}@\mathsf{ctxt.Coord}; \mathsf{Issuer}{=}@\mathsf{ctxt.Issuer}] \quad \mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E}', \epsilon, \{\ell_1\}) \quad \text{fresh } \ell 1, \ell 2 \quad \mathsf{FS}'{=}\mathsf{FS}{\cup}\{\ell 1, \ell 2\} \quad \text{fresh } \mathsf{Var}_m \quad \mathsf{Var}_m[\text{this}{\mapsto}(\ell 1, \ell_2), \arg{\mapsto}(\ell 2, \mathsf{null})] \quad c_m{=}\mathsf{MethodCmds}(\mathsf{TypeOf}(v), m) \quad \mathsf{E}'[\mathsf{Var}{=}\mathsf{Var}_m; \mathsf{FS}{=}\mathsf{FS}'; \mathsf{AM}{=}\mathsf{AM}_R] \vdash \boxed{c_m} \Rightarrow \mathsf{E}''}{\mathsf{E} \vdash \boxed{v.m(i_l)@\mathsf{ctxt}} \Rightarrow \mathsf{E}''[\mathsf{Var}{=}\mathsf{E.Var}]} \quad (\text{METHOD-CALL-ARG-DEFERRED})$$

$$\frac{[v{\mapsto}(\ell_1, val_v)]{\subseteq}\mathsf{Var} \quad \mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E}, \epsilon, \{\ell_1\})}{\mathsf{E} \vdash \boxed{v = \mathsf{null}} \Rightarrow \mathsf{E}[\mathsf{AM}{=}\mathsf{AM}_R]} \quad (\text{EQ})$$

Figure 10.1: Static Execution Rules (Part I).

$$\frac{\begin{array}{c}(\underline{\text{METHOD}-\text{CALL}-\text{ARG}-\text{NO}-\text{DEFER}}) \\ [v{\mapsto}(\ell_1,\ell_2)]{\subseteq}\text{Var} \quad \{\ell_1,\ell_2\}{\subseteq}\text{FS} \quad \ell_1{\neq}\ell_2 \quad \ell_2{\in}\text{Dom}(\text{Obj}) \quad \text{fresh } \ell 1,\ell 2 \\ \text{FS}'{=}\text{FS}{\cup}\{\ell 1,\ell 2\} \quad \text{fresh Var}_m \quad \text{AM}_R{=}\text{Add}_{\text{AR}}(\text{E},\epsilon,\{\ell_1\}) \\ \text{Var}_m[\text{this}{\mapsto}(\ell 1,\ell_2),\arg{\mapsto}(\ell 2,\text{null})] \quad c_m{=}\text{MethodCmds}(\text{TypeOf}(v),m) \\ \text{E}[\text{Var}{=}\text{Var}_m;\text{FS}{=}\text{FS}';\text{AM}{=}\text{AM}_R] \vdash c_m \Rightarrow \text{E}'\end{array}}{\text{E} \vdash v.m(i_l)@\text{nodefer} \Rightarrow \text{E}'[\text{Var}{=}\text{E.Var}]}$$

$$\frac{\text{E}[\text{Issuer}{=}\text{id},\text{Coord}{=}\mathcal{A}] \vdash c \Rightarrow \text{E}'[\text{Issuer}{=}\bot,\text{Coord}{=}\bot]}{\text{E} \vdash \text{id:atomic}\{c\} \Rightarrow \text{E}'}\ (\underline{\text{TRANSACTION}})$$

$$\frac{\begin{array}{c}[v{\mapsto}(\ell_1,val_v)]{\subseteq}\text{Var} \quad \ell_1{\in}\text{FS} \\ \text{E}[\text{Issuer}{=}\text{id};\text{Coord}{=}\mathcal{L}(\ell_1)] \vdash c \Rightarrow \text{E}'[\text{Issuer}{=}\bot;\text{Coord}{=}\bot]\end{array}}{\text{E} \vdash \text{id:sync}(v)\{c\} \Rightarrow \text{E}'}\ (\underline{\text{LOCK}})$$

$$\frac{\text{E} \vdash b \Rightarrow \text{E}' \quad \text{AM}_R{=}\text{Add}_{\text{AR}}(\text{E}',\epsilon,fn) \quad \text{E}'[\text{AM}{=}\text{AM}_R] \vdash c_m \Rightarrow \text{E}''}{\text{E} \vdash @\text{iter}{-}\text{space}[fn]\ \text{while } b\ \{c_m\} \Rightarrow \text{E}''}\ (\underline{\text{WHILE}})$$

$$\frac{\begin{array}{c}[v{\mapsto}(\ell_1,\ell_2),x{\mapsto}(\ell_3,\ell_4)]{\subseteq}\text{Var} \quad \ell_1{\neq}\ell_2 \quad \ell_3{\neq}\ell_4 \quad \ell_1{\neq}\ell_4 \quad \ell_3{\neq}\ell_2 \\ \{\ell_2,\ell_4\}{\subseteq}\text{Dom}(\text{Obj}) \quad \ell_{vf}{=}\text{FldLoc}(\text{E},v,f) \quad \{\ell_1,\ell_2,\ell_3,\ell_4,\ell_{vf}\} \subseteq \text{FS} \\ \text{AM}_R{=}\text{Add}_{\text{AR}}(\text{E},\epsilon,\{\ell_1,\ell_3\}) \quad \text{AM}_W{=}\text{Add}_{\text{AR}}(\text{E},1,\{\ell_{vf}\}) \\ \text{AM}'{=}\text{MergeAMs}(\text{AM}_R,\text{AM}_W) \quad \text{Obj}'{=}\text{FldUpd}(\text{E},v,f,\ell_4)\end{array}}{\text{E} \vdash v.f := x \Rightarrow \text{E}[\text{Obj}{=}\text{Obj}';\text{AM}{=}\text{AM}']}$$

$$\frac{[v{\mapsto}(\ell_1,\text{null})]{\subseteq}\text{Var} \quad \ell_1{\in}\text{FS} \quad \text{AM}_W{=}\text{Add}_{\text{AR}}(\text{E},1,\{\ell_1\})}{\text{E} \vdash v := i_l \Rightarrow \text{E}[\text{AM}{=}\text{AM}_W]}\ (\underline{\text{ASSIGN}-\text{INT}-\text{LITERAL}})$$

Figure 10.2: Static Execution Rules (Part II)

$$\frac{\begin{array}{c}(\underline{\text{FLD}-\text{UPDATE}-\text{FLD}-\text{REF}})\\ [v{\mapsto}(\ell_1,\ell_2),x{\mapsto}(\ell_3,\ell_4)]{\subseteq}\mathsf{Var}\quad \ell_{vf}{=}\mathsf{FldLoc}(\mathsf{E},v,f)\quad \ell_{xf}{=}\mathsf{FldLoc}(\mathsf{E},x,f)\\ \{\ell_1,\ell_2,\ell_3,\ell_4,\ell_{vf},\ell_{xf}\}\subseteq\mathsf{FS}\quad \ell_1{\neq}\ell_2\quad \ell_3{\neq}\ell_4\quad \ell_1{\neq}\ell_4\quad \ell_3{\neq}\ell_2\\ \{\ell_2,\ell_4\}\subseteq\mathsf{Dom}(\mathsf{Obj})\quad \mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E},\epsilon,\{\ell_1,\ell_3,\ell_{xf}\})\\ \mathsf{AM}_W{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E},1,\{\ell_{vf}\})\quad \mathsf{Obj}'{=}\mathsf{FldUpd}(\mathsf{E},v,f,\mathsf{FldVal}(\mathsf{E},x,f))\\ \mathsf{AM}'{=}\mathsf{MergeAMs}(\mathsf{AM}_R,\mathsf{AM}_W)\end{array}}{\mathsf{E}\vdash\ v.f:=x.f\ \Rightarrow \mathsf{E}[\mathsf{Obj}{=}\mathsf{Obj}';\mathsf{AM}{=}\mathsf{AM}']}$$

$$\frac{\begin{array}{c}(\underline{\text{ASSIGN}-\text{FLD}-\text{REF}})\\ [v{\mapsto}(\ell_1,val_v),x{\mapsto}(\ell_2,\ell_3)]{\subseteq}\mathsf{Var}\quad \ell_{xf}{=}\mathsf{FldLoc}(\mathsf{E},x,f)\quad \{\ell_1,\ell_2,\ell_3,\ell_{xf}\}\subseteq\mathsf{FS}\\ \ell_1{\neq}\ell_3\quad \ell_2{\neq}\ell_3\quad \ell_3\in\mathsf{Dom}(\mathsf{Obj})\\ \mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E},\epsilon,\{\ell_2,\ell_{xf}\})\quad \mathsf{AM}_W{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E},1,\{\ell_1\})\\ \mathsf{AM}'{=}\mathsf{MergeAMs}(\mathsf{AM}_R,\mathsf{AM}_W)\quad \mathsf{Var}'{=}\mathsf{Var}[v{\mapsto}(\ell_1,\mathsf{FldVal}(\mathsf{E},x,f))]\end{array}}{\mathsf{E}\vdash\ v:=x.f\ \Rightarrow \mathsf{E}[\mathsf{Var}{=}\mathsf{Var}';\mathsf{AM}{=}\mathsf{AM}']}$$

$$\frac{\begin{array}{c}(\underline{\text{FLD}-\text{UPDATE}-\text{VAR}-\text{LITERAL}})\\ [v{\mapsto}(\ell_1,\ell_2),x{\mapsto}(\ell_3,\mathsf{null})]{\subseteq}\mathsf{Var}\quad \ell_{vf}{=}\mathsf{FldLoc}(\mathsf{E},v,f)\quad \ell_1{\neq}\ell_2\quad \ell_3{\neq}\ell_2\\ \ell_2{\in}\mathsf{Dom}(\mathsf{Obj})\quad \{\ell_1,\ell_2,\ell_3,\ell_{vf}\}{\subseteq}\mathsf{FS}\quad \mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E},\epsilon,\{\ell_1,\ell_3\})\\ \mathsf{AM}_W{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E},1,\{\ell_{vf}\})\quad \mathsf{AM}'{=}\mathsf{MergeAMs}(\mathsf{AM}_R,\mathsf{AM}_W)\end{array}}{\mathsf{E}\vdash\ v.f:=x\ \Rightarrow \mathsf{E}[\mathsf{AM}{=}\mathsf{AM}']}$$

$$\frac{\begin{array}{c}(\underline{\text{PRINT}})\\ \mathsf{CheckSafeIO}(\mathsf{E})\\ [v{\mapsto}(\ell_1,\ell_2)]{\subseteq}\mathsf{Var}\quad \ell_{vf}{=}\mathsf{FldLoc}(\mathsf{E},v,f)\quad \ell_1{\neq}\ell_2\quad \{\ell_1,\ell_2,\ell_{vf}\}{\subseteq}\mathsf{FS}\\ \ell_2{\in}\mathsf{Dom}(\mathsf{Obj})\quad \mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E},\epsilon,\{\ell_1,\ell_{vf}\})\end{array}}{\mathsf{E}\vdash\ \mathsf{print}(v.f)\ \Rightarrow \mathsf{E}[\mathsf{AM}{=}\mathsf{AM}_R]}$$

$$\frac{\begin{array}{c}(\underline{\text{NEQ}})\\ [v{\mapsto}(\ell_1,val_v)]{\subseteq}\mathsf{Var}\quad \mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E},\epsilon,\{\ell_1\})\end{array}}{\mathsf{E}\vdash\ v\neq\mathsf{null}\ \Rightarrow \mathsf{E}[\mathsf{AM}{=}\mathsf{AM}_R]}$$

Figure 10.3: Static Execution Rules (Part III).

$$\frac{\begin{array}{c}[v{\mapsto}(\ell_1,\ell_2)]{\subseteq}\mathsf{Var}\quad\{\ell_1,\ell_2\}{\subseteq}\mathsf{FS}\quad\ell_1{\neq}\ell_2\quad\ell_2{\in}\mathsf{Dom}(\mathsf{Obj})\\ \mathsf{E}'{=}\mathsf{E}[\mathsf{TID}{=}@\mathsf{ctxt}.\mathsf{TID};\mathsf{Coord}{=}@\mathsf{ctxt}.\mathsf{Coord};\mathsf{Issuer}{=}@\mathsf{ctxt}.\mathsf{Issuer}]\\ \mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E}',\epsilon,\{\ell_1\})\quad\text{fresh }\ell 1\quad\mathsf{FS}'{=}\mathsf{FS}{\cup}\{\ell 1\}\quad\text{fresh }\mathsf{Var}_m\\ \mathsf{Var}_m[\mathsf{this}{\mapsto}(\ell 1,\ell_2)]\quad c_m{=}\mathsf{MethodCmds}(\mathsf{TypeOf}(v),m)\\ \mathsf{E}'[\mathsf{Var}{=}\mathsf{Var}_m;\mathsf{FS}{=}\mathsf{FS}';\mathsf{AM}{=}\mathsf{AM}_R]\vdash c_m\Rightarrow\mathsf{E}''\end{array}}{\mathsf{E}\vdash v.m()@\mathsf{ctxt}\Rightarrow\mathsf{E}''[\mathsf{Var}{=}\mathsf{E}.\mathsf{Var}]}\ (\mathsf{METHOD{-}CALL{-}NO{-}ARG{-}DEFERRED})$$

$$\frac{\begin{array}{c}[v{\mapsto}(\ell_1,\ell_2)]{\subseteq}\mathsf{Var}\quad\{\ell_1,\ell_2\}{\subseteq}\mathsf{FS}\quad\ell_1{\neq}\ell_2\quad\ell_2{\in}\mathsf{Dom}(\mathsf{Obj})\quad\text{fresh }\ell 1\\ \mathsf{FS}'{=}\mathsf{FS}{\cup}\{\ell 1\}\quad\text{fresh }\mathsf{Var}_m\quad\mathsf{AM}_R{=}\mathsf{Add}_{\mathsf{AR}}(\mathsf{E},\epsilon,\{\ell_1\})\\ \mathsf{Var}_m[\mathsf{this}{\mapsto}(\ell 1,\ell_2)]\quad c_m{=}\mathsf{MethodCmds}(\mathsf{TypeOf}(v),m)\\ \mathsf{E}[\mathsf{Var}{=}\mathsf{Var}_m;\mathsf{FS}{=}\mathsf{FS}';\mathsf{AM}{=}\mathsf{AM}_R]\vdash c_m\Rightarrow\mathsf{E}'\end{array}}{\mathsf{E}\vdash v.m()@\mathsf{nodefer}\Rightarrow\mathsf{E}'[\mathsf{Var}{=}\mathsf{E}.\mathsf{Var}]}\ (\mathsf{METHOD{-}CALL{-}NO{-}ARG{-}NO{-}DEFER})$$

$$\frac{\mathsf{E}\vdash c\Rightarrow\mathsf{E}'\qquad\mathsf{E}'\vdash c'\Rightarrow\mathsf{E}''}{\mathsf{E}\vdash c;c'\Rightarrow\mathsf{E}''}\ (\mathsf{SEQ_c{-}1})\qquad\frac{\mathsf{E}\vdash c\Rightarrow\mathsf{E}'}{\mathsf{E}\vdash c;\bullet\Rightarrow\mathsf{E}'}\ (\mathsf{SEQ_c{-}2})$$

$$\frac{\mathsf{E}\vdash c_m\Rightarrow\mathsf{E}'\qquad\mathsf{E}'\vdash c'_m\Rightarrow\mathsf{E}''}{\mathsf{E}\vdash c_m;c'_m\Rightarrow\mathsf{E}''}\ (\mathsf{SEQ_{c_m}{-}1})\qquad\frac{\mathsf{E}\vdash c_m\Rightarrow\mathsf{E}'}{\mathsf{E}\vdash c_m;\bullet\Rightarrow\mathsf{E}'}\ (\mathsf{SEQ_{c_m}{-}2})$$

$$\frac{\begin{array}{c}\mathsf{E}{=}0;\{\};\bot;\bot;\text{fresh }\mathsf{Var}_p;\text{fresh }\mathsf{Obj}_p;\text{fresh }\mathsf{AM}_p;\text{fresh }\mathsf{Dfr}\\ \mathsf{E}\vdash\text{Class-Decl}^*\Rightarrow\mathsf{E}\qquad\mathsf{E}\vdash(\text{Type }v)^+\Rightarrow\mathsf{E}'\\ \mathsf{E}'\vdash(v:=\text{new }cn\mid v:=i_l\mid v.f:=i_l)^+\Rightarrow\mathsf{E}''\\ \mathsf{E}''\vdash(v.m(i_l?)@\mathsf{ctxt})^*\Rightarrow\mathsf{E}'''\qquad\mathsf{E}_1{=}\mathsf{E}'''[\mathsf{AM}{=}[\,]]\\ \mathsf{E}_1[\mathsf{TID}{=}1]\vdash C_1\Rightarrow\mathsf{E}_2\qquad\ldots\qquad\mathsf{E}_n[\mathsf{TID}{=}n]\vdash C_n\Rightarrow\mathsf{E}_{n+1}\\ \mathsf{E}_m{=}\mathsf{E}_{n+1}\quad\text{serialised}{=}\mathsf{Serialise}(\mathsf{E}_m.\mathsf{Dfr})\quad\mathsf{E}_m\vdash\text{serialised}\Rightarrow\mathsf{E}'_m\quad\mathsf{E}_{\text{fin}}{=}\mathsf{E}'_m\end{array}}{\begin{array}{c}\mathsf{E}\vdash\\ \text{Class-Decl}^*\ (\text{Type }v)^+\ (v:=\text{new }cn\mid v:=i_l\mid v.f:=i_l)^+\\ (v.m(i_l?)@\mathsf{ctxt})^*\ (C_1||\ldots||C_n)\\ \Rightarrow\mathsf{E}_{\text{fin}}\end{array}}\ (\mathsf{PROGRAM})$$

Figure 10.4: Static Execution Rules (Part IV).

(VAR−DECL) executes a variable declaration:

- A fresh memory location ℓ is allocated (the variable's stack slot location).
- The memory location ℓ becomes bound in the program's free store FS'.
- v is associated with ℓ and the value null in Var'.

(ASSIGN−VAR−LITERAL) assigns the value of one variable to another, where both variables hold literal values:

- The memory locations associated with each variable must be bound in the program's free store.
- Executing the assignment results in a read access requirement on `x` and respectively write access requirement on `v`, each of which are contained in AM_R and respectively AM_W.
- AM' comprises the read and write contained in AM_R and AM_W as a result of merging AM_R and AM_W.

(NEW) executes an object allocation:

- The receiver of the allocation must have a literal value prior to the allocation. A variable can only be the recipient of a memory location once in the lifetime of a program unless the variable is declared within a method. (Methods are used to perform arbitrary assignment as we can reason about them in a simple uniform manner.)
- Execution of the allocation results in:

- A write access requirement on $\mathtt{v}$.
- Creation of the object mapping obj which represents the fields within the type cn. Where, $\mathsf{CreateObject} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Type} \rightarrow \mathsf{Object} \times \mathsf{LocationSet}$ returns a pair whose first component is the object obj modelling an instance of cn and second component the set of memory locations used by the fields in obj.
- The memory locations entailed by obj become bound in the program's free store.
- The base location of obj is the head of $locs$, where $\mathsf{Head}(\{\ell_1, \ldots, \ell_n\}) = \ell_1$.
- Var' updates the value of $\mathtt{v}$ to be the base location of obj.
- Obj' maps the base location of obj to obj.

(METHOD−CALL−DEFER) is applied to every method call issued with the parallel composition of threads.

- The method call is *dehydrated* by annotating the method call with a calling context, @ctxt, which states:
  - The thread identifier of the original method call.
  - The coordination type the method was originally executed under.
  - The issuing identifier, if applicable, of the coordination instance the method was invoked by.
- The dehydrated method call is appended to the list Dfr via ::, where $i :: [\,] = [i]$, $i' :: [i] = [i', i]$, and so on. The role of Dfr will be looked at when we describe (PROGRAM).

(METHOD−CALL−ARG−DEFERRED) executes a dehydrated method which takes an argument:

- The receiver of the method call must hold a value which is the base location of an object.
- The assertion $\ell_1 \neq \ell_2$ denotes that the memory in the stack and heap domains are distinct.
- E', the environment the method m is to be executed under, has its TID, Coord and Issuer components set to the values of the dehydrated method's TID, Coord and Issuer properties of its @ctxt annotation.
- The method call sees a read access requirement on the receiver `v`.
- Before the method can execute we create a method local variables mapping Var_m which has comprises *this* and *arg* (a metavariable over the method's formal argument) pushed in. Both *this* and *arg* are associated with fresh memory locations, and *this* takes on the value ℓ_2 which is the base location of the object the method is being invoked upon. This permits the method's program text to write or read the object's state, e.g. `this.field := ...`, and so on.
- The program text of the method is recalled via MethodCmds which takes the receiver type and a method name and returns the program text of that method. We assume this information is easily derivable from the program text.

- The program text of the invoked method is executed under an environment which uses Var_m. Upon completion of the method execution the method's variable mapping is swapped out for the global variable mapping.

(EQ) executes an equality check, which results in a read access requirement on v. (NEQ) (Figure 10.3) is the same but for an inequality check.

(METHOD−CALL−ARG−NO−DEFER) execute a method issued by the main thread. Each method issued by the main thread is annotated with @nodefer. (METHOD−CALL−ARG−NO−DEFER) is identical to (METHOD−CALL−ARG−DEFERRED) with the exception that the method's program text is executed under the present environment that differs only in its variable mapping. By contrast, (METHOD−CALL−ARG−DEFERRED) executes a method's program text under an environment whose TID, Coord and Issuer values are drawn from the dehydrated method's @ctxt annotation.

(TRANSACTION) executes a transaction. This entails setting the environment's Issuer component to the label id of the transactional instance and the Coord component to (TRANSACTION). Upon execution of the transactional commands the environment's Issuer and Coord components are both set to $\perp$. Executing a lock via (LOCK) is similar to (TRANSACTION) but the environment's Coord component is set to $\mathcal{L}(\ell_1)$, where ℓ_1 is the memory location associated with the variable being used as the mutex. Note that $\mathcal{L}$ is only parameterised on a memory location, by contrast to Part I where $\mathcal{L}$ was parameterised on a memory location and handle count.

(WHILE) executes a while loop:

- The access requirements issued by the boolean expression b are determined.

- The memory function fn is applied. A read access requirement is issued to each of the memory locations it returns. Note that fn is drawn from the while loop's @iter−space annotation.

- The body of the while loop is executed.

For example, the memory function of the while loop for `traverse` (Figure 9.2) is set to @object−space.dynamic which resolves to $\mathsf{nodes}(\mathsf{E}, \mathsf{head})$. When encountered in the program text The expression $\mathsf{nodes}(\mathsf{E}, \mathsf{head})$ is interpreted as $\mathsf{nodes}(\mathsf{E}, \mathsf{FldVal}(\mathsf{E}, \mathsf{this}, \mathsf{head}))$, which returns the set of memory locations that the `LinkedList` object referred to by *this* owns.

(FLD−UPDATE−VAR−REF) assigns the value of a variable holding a reference to an object to a field.

- Execution of the field update results in a read access requirement on `v` and `x`, and a write access requirement on `f`. The memory location of the field `f` in the indirection `v.f` is attained via $\mathsf{FldLoc} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \times \mathsf{Field} \rightarrow \mathsf{Location}$.

- The object mapping Obj' has the value of `f` being that of `x`'s value. The update is performed by $\mathsf{FldUpd} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \times \mathsf{FieldLocation} \rightarrow \mathsf{Obj}$ which returns an object mapping that is the same as $\mathsf{E.Obj}$ but differs in that the value of the field `f` of the relevant object has been updated to be $\ell 4$.

(ASSIGN−INT−LITERAL) is applied when assigning a literal value to a variable that currently holds a literal value. Its execution results in a write access requirement issued on `v`.

(PRINT) executes a print command:

- The predicate $\mathsf{CheckSafeIO} \stackrel{\mathsf{def}}{=} \mathsf{E} \rightarrow \mathsf{Bool}$ the environment is not in a transactional state. If the environment is in a transactional state then executing the command `print` may result in an inconsistent memory (Section 1.1.7.4).

- Executing the print command results in a read access requirement issued on `v` and `f`.

The remaining rules, apart from (PROGRAM), in Figures 10.3 and 10.4 are similar in operation to those previously described. Note that in the sequencing rules we use $\bullet$ to represent the empty command. (Typically this is ϵ but we have already used this to denote a read value.)

(PROGRAM) drives execution of a program.

- The environmental components are each initialised to their default values.

- The commands of the main thread are executed. Special attention should be drawn to E_1 throwing away the access requirements issued by the main thread – only access requirements issued by the parallel composition of threads is of importance.

- The commands of the first thread are executed yielding an environment which the commands of the second thread are executed from, and so on.

- The methods that were deferred while executing the parallel composition are serialised according to their respective class's @serialise definition. Serialise sorts the method invocations on each object according to the comparator specified by @serialise and returns the sorted method invocations as a command sequence. For example, Serialise(`o.traverse()`@ctxt,

`o.add(1)`@ctxt)=`o.add(1)`@ctxt; `l.traverse()`@ctxt, due to `add` < `traverse` in `LinkedList`@serialise. The serialised methods are then executed.

10.2 Isolation Algorithm

In this section we describe the predicate Isolated? $\stackrel{\text{def}}{=}$ AM $\rightarrow$ Bool given in Figure 10.5. Isolated? is given an access mapping instance and returns true if and only if the access requirements issued to each memory location in the domain of the supplied access mapping are isolated. Isolated? comprises of four general cases, each denoted with a label **C** and a short description. Isolated? can determine the DRF of any correctly annotated program using the facilities presented in Chapter 8, with the exception of when writes are issued to arbitrary locations of dynamically allocated memory, e.g. the middle of a linked list. This is not necessarily a limitation of the algorithm, but of the static execution rules themselves.

10.2.1 Preliminaries

Throughout the definition of Isolated? we use comments to describe the general or particular instance the case matches. To describe these cases accurately we use *access positions* and *access position modifiers*. Access positions denote whether or not an access of an memory location ℓ is in read, write or read/write position. An access position modifier wraps an access position to state the coordination type an access was issued under. An access position with no access position modifier is uncoordinated. Access positions are as follows:

- `:=` ℓ denotes ℓ is in read position;

- `ℓ :=` denotes ℓ is in write position; and
- `ℓ` denotes ℓ is in read/write position.

Access position modifiers include:

- `Coord(ℓ)` denotes the access of ℓ is issued by either a lock or transaction; and
- `Any(ℓ)` denotes the access of ℓ is issued uncoordinated, by a lock or by a transaction.

The definitions of the auxiliary functions referenced by Isolated? can be found in Appendix A.4.

10.2.2 Soundness of Isolation Algorithm

The function LocksAndTxnsIsolated (defined in Section A.4.11) implements the semantics described by Definition 7.3.

Theorem 10.1 (Isolation of Accesses)**.** *Let am be an access mapping* AM *derived from statically executing a program, such that* $\ell \in \mathsf{Dom}(am)$ *and* $|\mathsf{Dom}(am)| = 1$*. If* Isolated?(am) *then the accesses issued to* ℓ *are isolated and by extension DRF.*

Proof. The proof is structured over the cases of Figure 10.5.

- **C1**: if ℓ is only read, `Any(:= ℓ) || Any(:= ℓ)`, then due to Definition 7.6 accesses to ℓ are DRF; or, if ℓ is only accessed by a single thread, `ℓ || ...`, then accesses to ℓ are trivially isolated and DRF due to Definitions 7.2 and 7.6.

```
1: procedure Isolated?(am)
2:     for each ℓ ∈ Dom(am) do
3:         if NumberOfWritingThreads(am(ℓ)) = 0 ∨
               NumberOfAccessingThreads(am(ℓ)) = 1 then
4:             ▷ C1: ℓ only read or accessed by a single thread
5:             goto 2
6:         end if
7:         ▷ Several threads access ℓ; at least one is a write ————————————
8:         (un, txn, lk) ← PartitionAccessesByCoordType(am(ℓ))
9:         if un ≠ {} then
10:            ▷ C2: Uncoordinated accesses issued to ℓ
11:            if NumberOfWritingThreads(am(ℓ)) = 1 then
12:                writes ← Writes(am(ℓ))
13:                reads ← Reads(am(ℓ))
14:                writing_tid ← Head(writes).TID
15:                if ∃ar ∈ writes · ar.Coord=⊥ ∨
                       (∃ar ∈ reads · ar.Coord=⊥ ∧ ar.TID ≠ writing_tid) then
16:                    ▷ C2.1: ℓ := || Any(:= ℓ) or Coord(ℓ :=) || := ℓ
17:                    return False
18:                end if
19:                ▷ C2.2: Coord(ℓ :=) := ℓ || Coord(:= ℓ)
20:                if LocksAndTxnsIsolated(am, lk, txn) then goto 2
21:                else return False
22:                end if
23:            else
24:                ▷ C2.3: Any(ℓ :=) || Any(ℓ :=) || ℓ
25:                return False
26:            end if
27:        else if txn ≠ {} ∧ lk = {} then
28:            ▷ C3: All accesses issued to ℓ are transactional
29:            goto 2
30:        else
31:            ▷ C4: All lock, or lock and transactional accesses issued to ℓ
32:            if LocksAndTxnsIsolated(am, lk, txn) then goto 2
33:            else return False
34:            end if
35:        end if
36:    end for
37:    return True
38: end procedure
```

Figure 10.5: Isolation Algorithm

It follows from failing to satisfy **C1** that several threads access ℓ, with at least one of the accesses to ℓ being a write.

- **C2**: there exists an uncoordinated access issued to ℓ.
  - Assume a single thread writes ℓ.
    - **C2.1**: if an uncoordinated write is issued to ℓ, `ℓ:=||Any(:= ℓ)`, then due to Definitions 7.4 and 7.6 accesses to ℓ are not DRF. It follows from failing to satisfy the first part of the disjunct of **C2.1** that all writes to ℓ are coordinated. If there exists an uncoordinated read of ℓ issued outside of the writing thread, `Coord(ℓ :=) || := ℓ`, then due to Definitions 7.4 and 7.6 accesses to ℓ are not DRF.
    - **C2.2**: it follows from failing to satisfy case **C2.1** that all uncoordinated reads of ℓ are issued by the writing thread, that all writes of ℓ are coordinated and that the reads issued to ℓ outside of the writing thread are coordinated, `Coord(ℓ :=) :=ℓ||Coord(:=ℓ)`. Due to Definitions 7.2 and 7.6 the uncoordinated reads of ℓ issued by the writing thread are trivially isolated with the writing thread's writes of ℓ. Due to Definition 7.6 it follows that accesses to ℓ are DRF if and only if the coordinated writes of ℓ issued by the writing thread are isolated with respect to the coordinated reads of ℓ issued outside of the writing thread as defined by Definition 7.3.
  - Assume several threads write ℓ, `Any(ℓ :=) || Any(ℓ :=) || ℓ`. Due to Definitions 7.4 and 7.6 accesses to ℓ are not isolated.

If follows from failing to satisfy **C2** that accesses issued to ℓ are either issued transactionally, by locks or by locks and transactions.

- **C3**: if all accesses to ℓ are transactional, `atomic{`ℓ`} || atomic{`ℓ`}`, then accesses to ℓ are isolated due to Definitions 7.3 and 7.6.

If follows from failing to satisfy **C3** that all accesses issued to ℓ are either (i) issued by locks, or (ii) locks and transactions. **C4** covers both (i) and (ii).

- **C4**, (i) all accesses to ℓ are lock issued, `sync(`ℓ_1`){`ℓ`} || sync(`ℓ_2`){`ℓ`}`. Due to Definition 7.3 we require $\ell_1 = \ell_2$ for accesses to ℓ to be isolated and by extension DRF (Definition 7.6); otherwise, accesses to ℓ are not DRF.
- **C4**, (ii) accesses to ℓ are issued by locks and transactions, `sync(`ℓ_1`){`ℓ`} || atomic{`ℓ`;`ℓ_2`;}`. Due to Definition 7.3 each transactional instance that accesses ℓ must access the memory location used to protect the lock issued accesses of ℓ for the accesses to be isolated and by extension DRF (Definition 7.6). That is, the following must be true $\ell_1 = \ell_2$. Otherwise, accesses to ℓ are not DRF.

□

Theorem 10.2 (Program Isolation)**.** *Let am be the access mapping derived from statically executing a program prog. If* Isolated?(am)*, then prog is DRF.*

Proof. Trivial. Due to the structure of Isolated? it must be the case that Trivial 10.1 holds for each $\ell \in \mathsf{Dom}(am)$ for Isolated?(am). □

□

Isolated? is sufficient to determine the DRF of programs which issue accesses under locks, transactions, an uncoordinated semantics or some combination of those semantics. The main restriction of the algorithm is a consequence of the sensitivity of our static execution rules which collect access information. That is, while the algorithm can detect the DRF of a program which entails the previously mentioned access semantics, it cannot (with the information provided by the current static execution rules) determine the DRF of a program that entails operations which write an arbitrary location of dynamically allocated memory, e.g. writing in the middle of a linked list. Under the presented framework such a write would conservatively require that all accesses to the linked list be isolated w.r.t. the write, irrespective of whether the accesses logically conflicted. In effect, the algorithm would fall back to treating accesses in such a situation as being object rather than location-based, resulting in a conservative judgement. Additionally, the algorithm has no notion of specialised lock types, e.g. read/write locks or arbitrary semaphores. However, incorporating the latter semantics in Isolated? would be generally straightforward. Despite these deficiencies, Isolated? is more than capable of determining when accesses must be isolated in a transactional and lock-based setting (e.g., demanding that lock invariants are consistent, transactions access lock invariants only when required, and so on, when at least two threads access the same memory and one of them is a write), and when they need not be (e.g. single accessing thread or only readers). Examples of all instances Isolated? is applicable, along with derivations of their computation, are given in Appendix B.

10.3 Summary

In this chapter we have presented the static execution rules and isolation algorithm used by our static analysis framework. Application of each static execution rule results in a number of access requirements being issued to the memory allocated by a program. These accesses are maintained within the access mapping presented in Chapter 9. Upon completion of statically executing a program the access mapping is given to our isolation algorithm. The isolation algorithm applies a number of expert rules (the cases in Figure 10.5) to the accesses issued to each memory location. The expert rules are based upon the dynamic conflict semantics for locks and transactions covered in Part I of this thesis. If accesses to all the memory allocated by the program satisfy these expert rules then the program is judged to be isolated and by extension DRF.

We have presented a static framework for determining whether a program that uses locks, transactions or both to access shared memory is DRF. Our framework comprises two phases: static execution and application of our isolation algorithm. A program is statically executed to determine the memory it allocates. Memory is modelled by access requirements which are an enriched form of permission Boyland [2003]. An access requirement captures additional access metadata such as the issuing thread and the coordination semantics the access was issued under. The key advantage of access requirements is that they facilitate a simple and uniform means to determine the isolation of accesses, irrespective of the coordination semantics they were issued under. The access requirements a program issues during its static execution are captured by an access mapping. The access mapping maps each memory location to its set of access requirements. Our iso-

lation algorithm takes an access mapping that results from the static execution of a program and checks that all accesses are isolated. The isolation algorithm is capable of determining when lock, transactional and both lock and transactional accesses to the same memory are isolated. A program whose access mapping is deemed isolated by our isolation algorithm is DRF.

Chapter 11

Summary & Conclusions

We briefly summarise the contributions of this thesis and conclude by describing achieved results and possible future work.

11.1 Summary

In this thesis we have presented three contributions to aid reasoning about concurrent programs that use locks and transactions to issue accesses to the same memory: moverness for locks, transactions and guaranteed transactions; guaranteed transactions; and a static analysis framework for guaranteeing the data-race-freedom of programs entailing locks and transactions. Moverness is an abstract memory consistency model which distils the desired observation semantics for coordination tools into fours categories: left, right, both and free movers. We showed that moverness can be mapped to a memory consistency model such as Java's. Guaranteed transactions are an alternative in some cases to locks and the privatisation/publication idioms. The main advantage of guaranteed transactions is that they maintain a transactional interface and as such are easier to apply than locks or transactions when wishing to perform irreversible operations on shared data. We showed this by applying guaranteed transactions to

a scenario where a suffix of a list is serialised out to disk. Finally, we gave a static analysis framework for guaranteeing the data-race-freedom of a program that uses both locks and transactions to access the same memory. Mixing locks and transactions is error prone but provides the programmer greater flexibility when deciding which coordination semantics to issue accesses. We showed that our static analysis is sufficient for identifying data races in programs which issue accesses to non-dynamic and dynamic data structures.

11.2 Conclusions

11.2.1 Achieved Results

The objective of this thesis was to research techniques for reasoning about imperative concurrent programs which used both locks Dijkstra [1983] and transactions Shavit and Touitou [1995] to issue accesses to the same memory. The work in this thesis met this objective by presenting the following: *moverness* (see Chapter 5 and Barnett and Qin [2012a]) – an abstraction over write observation semantics; *guaranteed transactions* (see Chapter 6 and Barnett and Qin [2012b]) – a partial abstraction over the privatisation/publication idioms; and a *static analysis* (see Part II and Barnett and Qin [2013]) for determining whether such programs are data-race-free.

11.2.1.1 Moverness

Moverness defines the observation semantics of writes issued to memory under no coordination, lock or transactional semantics. Moverness is an abstraction over a low-level memory consistency model, and is required to reason about the values a read observes in a concurrent program. Definitions of the moverness laws were

given along with a projection of moverness onto the Java memory consistency model. Moverness permits the programmer to reason about the writes that reads issued from locks and transactions will observe without invalidating the semantics of the memory consistency model it abstracts.

11.2.1.2 Guaranteed Transactions

In a purely transactional programming model a strong pessimistic semantics are required for executing irreversible operations. We gave such a semantics in the form of guaranteed transactions, while preserving a transactional interface. We described guaranteed transactions by giving an operational semantics for an imperative Java-like language. We also described the concurrent operational semantics of guaranteed transactions with respect to optimistic, weakly isolated, out-of-place transactions. Guaranteed transactions were found to be a suitable replacement for the privatisation/publication idioms under situations when the structure of the accessed data structures are well known.

11.2.1.3 Static Analysis

Understanding the execution semantics of a concurrent program before it is admitted to the execution environment is important, particularly when the programming model is complex, such as that which affords locks and transactions. Our static analysis guaranteed that programs that used locks and transactions to issue accesses to the same memory were data-race-free. We described our static analysis by defining a static semantics making use of fractional permissions for a Java-like language, and gave theorems showing that programs admitted to the execution environment are data-race-free.

11.2.2 Future Work

We describe two natural extensions of this thesis for future work: application of our approach to a different transactional semantics, and a static analysis based upon separation logic.

This thesis focused on transactions that were weakly isolated, optimistic and out-of-place. Other transactional semantics exist such as object based STMs Harris et al. [2010], those based on the linearizability memory consistency model Herlihy and Wing [1990]; Koskinen et al. [2010] and others under active investigation such as ISO-WG21 [2012] which will make use of the C++ memory consistency model Boehm and Adve [2008]. Changes in the underlying STM may result in some novel discoveries regarding the dynamic and static semantics presented in this thesis. We have shown that interest in STM is relatively active (see Barnett and Qin [2012a,b, 2013]) which suggests that such discoveries would be of interest to the research community.

The static analysis presented in this thesis was based upon fractional permissions Boyland [2003, 2010] which have shown considerable promise for reasoning about concurrent programs. Analyses based on separation logic Reynolds [2002] using fractional permissions have recently shown encouraging results Bornat et al. [2005]. An interesting area of research would be to encode the static analysis given in this thesis using separation logic and fractional permissions. We believe that such an analysis would be more expressive and of significant importance to the research community. It is possible that such an analysis could aid in the verification of other related research such as the use of the privatisation/publication idioms Lev and Maessen [2005]; Smaragdakis et al. [2007]; Spear et al. [2007];

Ziarek et al. [2008] which are particularly important should STM be adopted by mainstream imperative languages such as C++ ISO-WG21 [2012].

Appendix A

Algorithm Definitions

Algorithms are labelled with a *type signature* which describes the types of its arguments and return value. The form of a type signature is $A \stackrel{\mathsf{def}}{=} t_1 \times \cdots \times t_n \to t_{ret}$, where $t_1, \ldots, t_n$ are the types of the arguments A expects to be provided and t_{ret} is its return type.

A.1 Types

Before we present the algorithms used in Parts I and II we give a quick summary of all types used:

- $\mathsf{Int} \stackrel{\mathsf{def}}{=} \mathbb{N}$.
- Variable comprises all possible variables identifiers.
- VariableSet is a set of Variable.
- Field comprises all possible field identifiers.

- $\mathsf{FieldSet}$ is a set of Field.
- $\mathsf{ID} \stackrel{\mathsf{def}}{=} \mathsf{Int}$.
- IDSet is a set of ID.
- $\mathsf{Issuer} \stackrel{\mathsf{def}}{=} \mathsf{Int}$.
- $\mathsf{Time} \stackrel{\mathsf{def}}{=} \mathsf{Int}$.
- $\mathsf{TID} \stackrel{\mathsf{def}}{=} \mathsf{Int}$.
- TIDSet is a set of TID.
- $\mathsf{Location}$ comprises all possible memory locations ℓ and the nullary location null.
- $\mathsf{LocationSet}$ is a set of $\mathsf{Location}$.
- FS is a set of $\mathsf{Location}$.
- $\mathsf{MD} \stackrel{\mathsf{def}}{=} \mathsf{ID} \to \mathsf{MetaData}$ takes an identifier associated with a lock, transaction or guaranteed transaction and returns its respective metadata.
- MDSet is a set of MD.
- $\mathsf{MetaData} \stackrel{\mathsf{def}}{=} \mathsf{Time} \times \mathsf{Time} \times \mathsf{LocationSet} \times \mathsf{LocationSet} \times \mathsf{LocationSet} \times \mathsf{Coord}$.
- $\mathsf{Coord} \stackrel{\mathsf{def}}{=} \bot \mid \mathcal{A} \mid \mathcal{L} \mid \mathcal{G}$ is the union type comprising the values $\bot$, $\mathcal{A}$, $\mathcal{L}$ and $\mathcal{G}$ which represent no coordination semantics, transactions, locks and respectively guaranteed transactions. In Part I $\mathcal{L}$ is parameterised on two values: a memory location ℓ (the memory location of the mutex being used) and a handle count *count* which is an integer, $\mathcal{L}(\ell, \mathit{count})$. By contrast, in

Part II $\mathcal{L}$ is just parameterised on the memory location of a mutex, $\mathcal{L}(\ell)$. Also, in Part II Coord does not comprise the value $\mathcal{G}$.

- Store $\stackrel{\text{def}}{=}$ Variable $\rightarrow$ Location $\times$ Location takes a variable and returns a tuple whose first component is the location of the variable and the second its value.
- Heap $\stackrel{\text{def}}{=}$ Location $\rightarrow$ Object takes the base memory location of an object and returns the object to which it refers.
- Object $\stackrel{\text{def}}{=}$ FieldSet $\rightarrow$ Location $\times$ Location takes a field name and returns a tuple whose first component is the address of the field and second its value.
- Obj $\stackrel{\text{def}}{=}$ Location $\rightarrow$ Object takes the base memory location of an object and returns the object it refers to.
- State $\stackrel{\text{def}}{=}$ Store $\times$ Heap is a state which is a pair of store and heap mappings.
- StateSet is a set of State.
- Scale $\stackrel{\text{def}}{=}$ ϵ | 1 where ϵ is a read scale and 1 a write scale.
- AR $\stackrel{\text{def}}{=}$ TID $\times$ Scale $\times$ Coord $\times$ Issuer is an access requirement. Note that Coord $\stackrel{\text{def}}{=}$ $\perp$ | $\mathcal{L}$ | $\mathcal{A}$ and $\mathcal{L}$ is parameterised only on a memory location ℓ.
- ARSet is a set of AR
- Bool $\stackrel{\text{def}}{=}$ True | False.
- DeferredMethodCall contains all instances of the form $v.m(i_l?)$@ctxt.
- DeferredMethodCallList is a list of DeferredMethodCall.

- DeferredMethodCallSequence is a sequence of DeferredMethodCall delimited by ;.
- Type contains all user defined types.

A.2 Algorithm Definitions for Operational Semantics

A.2.1 Algorithms

A.2.1.1 GenerateID

GenerateID $\stackrel{\text{def}}{=}$ MD $\times$ ID $\rightarrow$ ID generates the next unique label not in the domain of the given metadata mapping.

$$\mathsf{GenerateID}(md, id) = id' \textbf{ where } id'{=}\mathsf{Succ}(id) \wedge id' \notin \mathsf{Dom}(md)$$

Where, Succ $\stackrel{\text{def}}{=}$ ID $\rightarrow$ ID gives the successor of the previously unique label, $\mathsf{Succ}(id) = id + 1$.

Example A.1 (GenerateID)**.** Let md be an instance of a metadata mapping MD such that $[1{\mapsto}(3, \perp, \{\}, \{\}, \{\}, \mathcal{A})]{\subseteq}md$ and id be a valid instance of ID such that $id = 1$. $\mathsf{GenerateID}(md, id) = 2$. $\square$

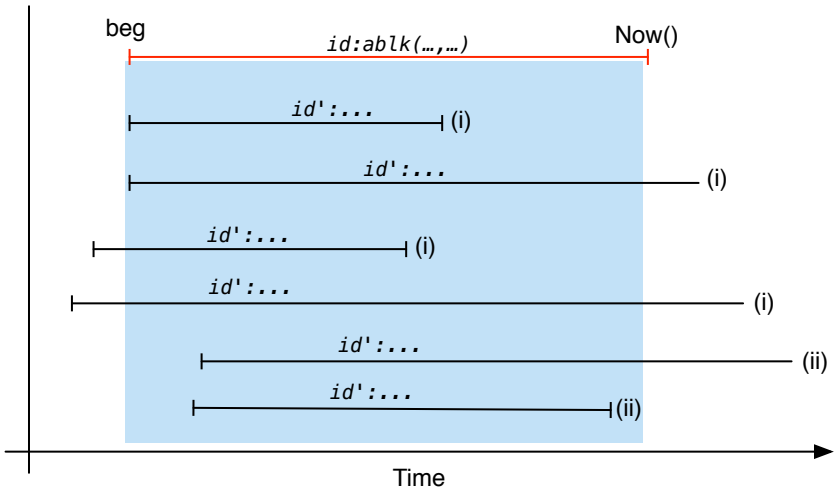
A.2.1.2 Conflict

Conflict $\stackrel{\text{def}}{=}$ ID $\times$ MD $\rightarrow$ Bool is a predicate that asserts whether the identifier of the transaction provided conflicts with another actively running transaction or

lock.

$$
\begin{aligned}
\mathsf{Conflict}(id, md) \stackrel{\mathsf{def}}{=}\ & \exists id' \neq id \in \mathsf{Dom}(md) \cdot \\
& [id \mapsto (\mathsf{beg}i, \bot, \gamma_{\mathsf{R}}i, \gamma_{\mathsf{W}}i, \gamma_{\mathsf{D}}i, \mathcal{A})] \subseteq md \wedge \\
& [id' \mapsto (\mathsf{beg}j, \mathsf{cmt}j, \gamma_{\mathsf{R}}j, \gamma_{\mathsf{W}}j, \gamma_{\mathsf{D}}j, \mathsf{coord})] \subseteq md \wedge \\
& \gamma_{\mathsf{D}}i \cap \gamma_{\mathsf{W}}j \neq \{\} \wedge \\
(i)\quad & ((\mathsf{beg}i \geq \mathsf{beg}j \wedge \\
(i)\quad & \qquad (\mathsf{cmt}j \leq \mathsf{Now}() \vee \mathsf{cmt}j = \bot)) \\
& \vee \\
(ii)\quad & (\mathsf{beg}i < \mathsf{beg}j \wedge \\
(ii)\quad & \qquad (\mathsf{cmt}j \leq \mathsf{Now}() \vee \mathsf{cmt}j = \bot)))
\end{aligned}
$$

Example A.2 (Conflict)**.** Consider the following diagram where the red interval represents the transaction instance i. Straight edges indicate the transaction is yet to commit. Intervals on a different line denote a particular case. The labels (i) and (ii) are used to denote which part of Conflict each respective interval matches against. ... denote either a lock or transaction.



□

A.2.1.3 Persist

Persist $\stackrel{\text{def}}{=}$ State × Store × State → Store × State given in Algorithm 1 takes a redo log, a thread store and a global state and returns an updated thread store and global state with the effect of the redo log persisted.

Algorithm 1 Persist $\stackrel{\text{def}}{=}$ State × Store × State → Store × State

```
1:  procedure Persist(redo,store,state)
2:      s_τ ← store
3:      σ ← state
4:      for each v ∈ Dom(redo.s) do
5:          if v ∈ Dom(s_τ) then
6:              s_τ ← s_τ[v↦redo.s(v)]
7:          else
8:              σ.s ← σ.s[v↦redo.s(v)]
9:          end if
10:     end for
11:     for each ℓ ∈ Dom(redo.h) do
12:         σ.h ← σ.h[ℓ↦redo.h(ℓ)]
13:     end for
14:     return (s_τ, σ)
15: end procedure
```

Example A.3 (Persist)**.** Let δ and σ be instances of State and s_τ be an instance of Store, such that

$$[v{\mapsto}(\ell 1, \ell 2), x{\mapsto}(\ell 3, \ell 4)]{\subseteq}\delta.\mathsf{s} \quad [\ell 2{\mapsto}o1, \ell 4{\mapsto}o2]{\subseteq}\delta.\mathsf{h}$$
$$[v{\mapsto}(\ell 1, \ell 5)]{\subseteq}\mathsf{s}_\tau \quad [x{\mapsto}(\ell 3, \ell 6)]{\subseteq}\sigma.\mathsf{s} \quad [\ell 2{\mapsto}o4, \ell 4{\mapsto}o5]{\subseteq}\sigma.\mathsf{h}$$

$(\mathsf{s}'_\tau, \sigma') = \mathsf{Persist}(\delta, \mathsf{s}_\tau, \sigma)$, where

$$[v{\mapsto}(\ell 1, \ell 2)]{\subseteq}\mathsf{s}'_\tau \quad [x{\mapsto}(\ell 3, \ell 4)]{\subseteq}\sigma'.\mathsf{s} \quad [\ell 2{\mapsto}o1, \ell 4{\mapsto}o2]{\subseteq}\sigma'.\mathsf{h} \quad \square$$

A.2.1.4 Acquireable

$\mathsf{Acquireable} \stackrel{\mathsf{def}}{=} \mathsf{Location} \times \mathsf{MD} \to \mathsf{Bool}$ is a predicate that asserts whether a currently active lock is still in pocession of the lock at the specified memory location.

$$\begin{aligned}\mathsf{Acquireable}(loc, md) \stackrel{\mathsf{def}}{=}\ & \nexists id{\in}\mathsf{Dom}(md) \cdot \\ & [id{\mapsto}(\mathsf{beg}, \bot, \{\}, \{\}, \{\ell\}, \mathcal{L})]{\subseteq}md \wedge \\ & \mathsf{beg} \neq \bot \wedge \ell{=}loc\end{aligned}$$

The predicate is relatively simple to digest. It states that *loc* is acquireable if and only if there does not exist an actively running lock that has already acquired *loc*. Note that Acquireable will be false if called by a child lock whose mutex is the same as that of its parent lock. This occurs because we use $\mathcal{L}$ without parameters to match all values of $\mathcal{L}$.

Example A.4 (Acquireable)**.** Let md be an instance of MD and ℓ be a valid memory location in the free store, such that $[id{\mapsto}(4, \bot, \{\}, \{\}, \{\ell\}, \mathcal{L})]{\subseteq}md$. $\mathsf{Acquireable}(\ell, md) = \mathsf{False}$. However, given $[id{\mapsto}(4, \bot, \{\ell\}, \{\}, \{\ell\}, \mathcal{A})]{\subseteq}md$ we

have $\mathsf{Acquireable}(\ell, md) = \mathsf{True}$. $\square$

The version of Acquireable we use in the parallel composition rule is given in Algorithm 2 and is similar to that given previously but encapsulates the computation of a variable's memory location.

Algorithm 2 $\mathsf{Acquireable} \stackrel{\mathsf{def}}{=} \mathsf{ID} \times \mathsf{MD} \times \mathsf{TID} \times \mathsf{Store} \times \mathsf{State} \times \mathsf{Variable} \to \mathsf{Bool}$

```
1: procedure Acquireable(id,md,tid,s,σ,v)
2:     ℓ ← VarLocation(s, σ, v)
3:     return ∄id′≠id ∈ Dom(md)·
4:                 [id′↦(beg, ⊥, {}, {}, {ℓ}, ℒ(τ≠tid, count))]⊆md
5: end procedure
```

Algorithm 2 asserts that a thread that differs to tid is not executing a lock that has acquired ℓ, the location of the mutex tid's lock wishes to acquire.

A.2.1.5 HeldByThread

$\mathsf{HeldByThread} \stackrel{\mathsf{def}}{=} \mathsf{TID} \times \mathsf{Location} \times \mathsf{MD} \to \mathsf{Bool}$ is a predicate that asserts that the mutex location specified is already held by the given thread.

$$\begin{aligned}\mathsf{HeldByThread}(tid, loc, md) \stackrel{\mathsf{def}}{=}\ & \exists id \in \mathsf{Dom}(md)\cdot \\ & [id\mapsto(\mathsf{beg}, \bot, \{\}, \{\}, \{loc\}, \mathcal{L}(tid, count))]\subseteq md \\ & \wedge\ count \geq 1\end{aligned}$$

Example A.5 (HeldByThread)**.** Let md be an instance of MD, 1 be a valid thread identifier in TID and ℓ a location in Location, where $[id\mapsto(2, \bot, \{\}, \{\}, \{\ell\}, \mathcal{L}(1, 1))]\subseteq md$. $\mathsf{HeldByThread}(1, \ell, md) = \mathsf{True}$. $\square$

A.2.1.6 VarLocation

$\mathsf{VarLocation} \stackrel{\mathsf{def}}{=} \mathsf{Store} \times \mathsf{State} \times \mathsf{Variable} \rightarrow \mathsf{Location}$ given in Algorithm 3 looks up the memory location of the specified variable identifier.

Algorithm 3 $\mathsf{VarLocation} \stackrel{\mathsf{def}}{=} \mathsf{Store} \times \mathsf{State} \times \mathsf{Variable} \rightarrow \mathsf{Location}$

```
1: procedure VarLocation(s, σ, v)
2:     loc ← null
3:     if ∃v ∈ Dom(s) then
4:         loc ← fst(s(v))
5:     else
6:         loc ← fst(σ.s(v))
7:     end if
8:     return loc
9: end procedure
```

Example A.6 (VarLocation)**.** Let s be an instance of Store, σ an instance of State and v an instance of $\mathsf{Variable}$, such that $[v \mapsto (\ell 1, \ell 2)] \subseteq \mathsf{s}$.

$\mathsf{VarLocation}(\mathsf{s}, \sigma, v) = \ell 1$. □

We also use $\mathsf{VarLocation}$ (Algorithm 4) in the unified rules where we have only a single store. Therefore, we provide the alternative $\mathsf{VarLocation} \stackrel{\mathsf{def}}{=} \mathsf{Store} \times \mathsf{Variable} \rightarrow \mathsf{Location}$.

Algorithm 4 $\mathsf{VarLocation} \stackrel{\mathsf{def}}{=} \mathsf{Store} \times \mathsf{Variable} \rightarrow \mathsf{Location}$

```
1: procedure VarLocation(s, v)
2:     loc ← fst(s(v))
3:     return loc
4: end procedure
```

Example A.7 (VarLocation)**.** Let s be an instance of Store and v an instance of $\mathsf{Variable}$, such that $[v \mapsto (\ell 1, \ell 2)] \subseteq \mathsf{s}$. $\mathsf{VarLocation}(\mathsf{s}, v) = \ell 1$. □

A.2.2 IsNull

The predicate $\mathsf{IsNull} \stackrel{\mathsf{def}}{=} \mathsf{Location} \to \mathsf{Bool}$ checks if a value is null.

$$\mathsf{IsNull}(\ell) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{True} & \text{if } \ell = \mathsf{null} \\ \mathsf{False} & \text{otherwise} \end{cases}$$

Example A.8 (IsNull)**.** $\mathsf{IsNull}(\mathsf{null}) = \mathsf{True}$. □

A.2.3 CreateObject

$\mathsf{CreateObject} \stackrel{\mathsf{def}}{=} \mathsf{Type} \times \mathsf{FS} \to \mathsf{Object} \times \mathsf{LocationSet}$ given in Algorithm 5 creates an object for a type cn. The first component of the returned tuple is an object $[f_1 \mapsto (\ell_1, \mathsf{null}), \ldots, f_n \mapsto (\ell_n, \mathsf{null})]$ where $\{f_1, \ldots, f_n\} \subseteq \mathsf{TypeFields}(cn)$; the second component is the set of memory locations $\{\ell_1, \ldots, \ell_n\}$ associated with the fields of the object. $\mathsf{TypeFields} \stackrel{\mathsf{def}}{=} \mathsf{Type} \to \mathsf{FieldSet}$ returns the set of fields a type comprises. We assume this information is derivable from the program text. We carry a FS instance to give context to fresh ℓ.

Algorithm 5 $\mathsf{CreateObject} \stackrel{\mathsf{def}}{=} \mathsf{Type} \times \mathsf{FS} \to \mathsf{Object} \times \mathsf{LocationSet}$

```
1:  procedure CreateObject(env, cn)
2:      obj ← fresh Object
3:      locs ← {}
4:      Dom(obj) ← TypeFields(cn)
5:      for each f ∈ Dom(obj) do
6:          ℓ_f ← fresh ℓ
7:          locs ← locs ∪ {ℓ_f}
8:          obj ← obj[f ↦ (ℓ_f, null)]
9:      end for
10:     return (obj, locs)
11: end procedure
```

Example A.9 (CreateObject)**.** Let fs be an instance of FS such that FS $=$ $\{\ell 1, \ell 2\}$. Further, let `class Point { Int x; Int y}`. CreateObject(fs, `Point`) $=$ $(obj, locs)$, where $[\mathsf{x}{\mapsto}(\ell 3, \mathsf{null}), \mathsf{y}{\mapsto}(\ell 4, \mathsf{null})]{\subseteq}obj$ and $locs{=}\{\ell 3, \ell 4\}$. □

Rather than define them twice, as they are almost identical, the functions FldLoc, FldVal and FldUpd are almost identical to those in Section A.3 with the exception that the first parameter is an instance of State.

A.2.4 PassByValue

PassByValue $\stackrel{\text{def}}{=}$ State $\times$ FS $\times$ Variable $\times$ VariableSet $\rightarrow$ Store $\times$ LocationSet given in Algorithm 6 copies the values of the variables given and returns a tuple whose first component is a store defined for the formal arguments of a method and second component the memory locations associated with the variables in the returned store. The first set of variables are the names of the actual variables passed to the method and the second set the names of the method's formal arguments. Where, $\mathsf{Zip}(\{a, b, c\}, \{1, 2, 3\}) = \{(a, 1), (b, 2), (c, 3)\}$.

Example A.10 (PassByValue)**.** Let σ be an instance of State and fs an instance of FS, such that $[v{\mapsto}(\ell 1, \ell 2), x{\mapsto}(\ell 3, \ell 4), y{\mapsto}(\ell 5, \ell 6)]{\subseteq}\sigma.\mathsf{s}$ and $\mathsf{fs}{=}\{\ell 1, \ell 2, \ell 3, \ell 4, \ell 5, \ell 6\}$. PassByValue$(\sigma, \mathsf{fs}, y, \{v, x\}, \{arg1, arg2\}){=}(\mathsf{s}_m, locs)$, where $[arg1{\mapsto}(\ell 7, \ell 2), arg2{\mapsto}(\ell 8, \ell 4), this{\mapsto}(\ell 9, \ell 6)]{\subseteq}\mathsf{s}_m$ and $locs{=}\{\ell 7, \ell 8, \ell 9\}$. □

A.2.5 ArgLocs

ArgLocs $\stackrel{\text{def}}{=}$ State $\times$ VariableSet $\rightarrow$ LocationSet given in Algorithm 7 returns the memory locations of the specified variables.

Algorithm 6 PassByValue $\stackrel{\text{def}}{=}$ State $\times$ FS $\times$ Variable $\times$ VariableSet $\times$ VariableSet $\rightarrow$ Store $\times$ LocationSet

```
 1: procedure PassByValue(σ, fs, receiver, vs, fvs)
 2:     s_m ← fresh Store
 3:     vars ← Zip(vs, fvs)
 4:     Dom(s_m) ← fvs
 5:     locs ← {}
 6:     for each v ∈ vars do
 7:         loc ← fresh ℓ
 8:         s_m ← s_m[snd(v)↦(loc, snd(σ.s(fst(v))))]
 9:         locs ← locs ∪ {loc}
10:     end for
11:     loc ← fresh ℓ
12:     locs ← locs ∪ {loc}
13:     Dom(s_m) ← Dom(s_m) ∪ {this}
14:     s_m ← s_m[this↦(loc, snd(σ.s(receiver)))]
15:     return (s_m, locs)
16: end procedure
```

Algorithm 7 ArgLocs $\stackrel{\text{def}}{=}$ State $\times$ VariableSet $\rightarrow$ LocationSet

```
procedure ArgLocs(σ, vs)
    locs ← {}
    for each v ∈ vs do
        locs ← locs ∪ {fst(σ.s(v))}
    end for
    return locs
end procedure
```

Example A.11 (ArgLocs)**.** Let σ be an instance of State such that

$[v\mapsto(\ell 1,\ell 2), x\mapsto(\ell_3,\ell_4)]\subseteq\sigma.\text{s}$. ArgLocs$(\sigma,\{v,x\})=\{\ell 1,\ell 3\}$. $\square$

A.2.6 GConflict

GConflict $\stackrel{\text{def}}{=}$ LocationSet $\times$ MD $\rightarrow$ Bool is a predicate that determines whether or not the write set of a guaranteed transaction conflicts with the dataset of an actively running guaranteed transaction. Because GConflict is pessimistic we

do not need to check for conflicts with intersecting execution intervals like we do in Conflict. Observe that a guaranteed transaction is free to execute if its write set conflicts with the dataset of an active transaction. Here, the guaranteed transaction will force the abortion of the transaction (see Conflict).

$$\begin{aligned}\mathsf{GConflict}(ws, \mathsf{md}) \stackrel{\mathsf{def}}{=}\ & \exists id \in \mathsf{Dom}(\mathsf{md}) \cdot \\ & [id \mapsto (\mathsf{beg}, \bot, \gamma_{\mathsf{R}}, \gamma_{\mathsf{W}}, \gamma_{\mathsf{D}}, \mathcal{G})] \subseteq \mathsf{md} \\ & \wedge\ ws \cap \gamma_{\mathsf{D}} \neq \{\}\end{aligned}$$

Example A.12 (GConflict)**.** Let md be an instance of MD such that $[1 \mapsto (4, \bot, \{\ell 1, \ell 2\}, \{\ell 3\}, \{\ell 1, \ell 2, \ell 3\}, \mathcal{G})] \subseteq \mathsf{md}$ and $ws = \{\ell 3, \ell 4\}$. $\mathsf{GConflict}(ws, \mathsf{md}) = \mathsf{True}$. □

The version of GConflict we use in the parallel composition rule is identical to that shown before but additionally encapsulates the computation of a command's write set, as shown in Algorithm 8.

Algorithm 8 $\mathsf{GConflict} \stackrel{\mathsf{def}}{=} C \times \mathsf{Store} \times \mathsf{State} \times \mathsf{MD} \rightarrow \mathsf{Bool}$

```
1: procedure GConflict(c,s,σ,md)
2:     ws ← Writes(c, s, σ)
3:     return GConflict(ws, md)
4: end procedure
```

A.2.7 MaxLabel

$\mathsf{MaxLabel} \stackrel{\mathsf{def}}{=} \mathsf{IDSet} \rightarrow \mathsf{ID}$ returns the largest identifier from the set of unique identifiers provided.

Example A.13 (MaxLabel)**.** $\mathsf{MaxLabel}(\{1, 4, 2\}) = 4$. □

A.2.8 MergeMetadata

$\mathsf{MergeMetadata} \stackrel{\mathsf{def}}{=} \mathsf{MDSet} \to \mathsf{MD}$ given in Algorithm 9 returns a metadata mapping whose domain and co-domain are the merge of the metadata map instances provided. Note that the metadata values in each mapping are always complete.

Algorithm 9 $\mathsf{MergeMetadata} \stackrel{\mathsf{def}}{=} \mathsf{MDSet} \to \mathsf{MD}$

```
1: procedure MergeMetadata(mds)
2:     merged ← fresh MD
3:     for each md ∈ mds do
4:         Dom(merged) ← Dom(merged) ∪ Dom(md)
5:     end for
6:     for each id ∈ Dom(merged) do
7:         for each md ∈ mds do
8:             if id ∈ Dom(md) then
9:                 merged ← merged[id↦md(id)]
10:                break
11:            end if
12:        end for
13:    end for
14:    return merged
15: end procedure
```

Example A.14 ($\mathsf{MergeMetadata}$)**.** Let md_i, md_j and md_k be instances of MD such that

$$[2 \mapsto (3, 4, \{\}, \{\ell 8\}, \{\ell 8\}, \mathcal{A})] \subseteq \mathsf{md}_i$$

$$[3 \mapsto (8, 12, \{\ell 1, \ell 2\}, \{\}, \{\ell 1, \ell 2\}, \mathcal{A})] \subseteq \mathsf{md}_j$$

$$[5 \mapsto (3, 9, \{\ell 3, \ell 4\}, \{\ell 5\}, \{\ell 3, \ell 4, \ell 5\}, \mathcal{A}), 2 \mapsto (3, 4, \{\}, \{\ell 8\}, \{\ell 8\}, \mathcal{A})] \subseteq \mathsf{md}_k$$

$\mathsf{MergeMetadata}(\{\mathsf{md}_i, \mathsf{md}_j, \mathsf{md}_k\}) = merged$, where

$$[2{\mapsto}(3, 4, \{\}, \{\ell 8\}, \{\ell 8\}, \mathcal{A}),$$
$$3{\mapsto}(8, 12, \{\ell 1, \ell 2\}, \{\}, \{\ell 1, \ell 2\}, \mathcal{A}),$$
$$5{\mapsto}(3, 9, \{\ell 3, \ell 4\}, \{\ell 5\}, \{\ell 3, \ell 4, \ell 5\}, \mathcal{A})]{\subseteq}merged$$

□

A.2.9 MergeStates

$\mathsf{MergeStates} \stackrel{\mathsf{def}}{=} \mathsf{StateSet} \rightarrow \mathsf{State}$ returns a state whose store and heap components are the merge of the domain and co-domains of the states provided. Note that MergeStates will overwrite the value of store and heap values when they differ. The value of the mappings in the formed state will only be as expected if the states provided are a product of a correctly coordinated reduction in the program semantics.

Example A.15 (MergeStates)**.** Let $\mathsf{s}_i, \mathsf{s}_j, \mathsf{h}_i$ and h_j be valid instances of Store and respectively Heap such that

$$[v{\mapsto}(\ell_1, \mathsf{null}), x{\mapsto}(\ell 3, \ell 4)]{\subseteq}\mathsf{s}_i \quad [v{\mapsto}(\ell_1, \ell 2), y{\mapsto}(\ell 5, \ell 6)]{\subseteq}\mathsf{s}_j$$
$$[\ell 5{\mapsto}[f_1{\mapsto}(\ell_5, \ell 6), f_2{\mapsto}(\ell 7, \ell 8)], \ell 9{\mapsto}[f_1{\mapsto}(\ell 9, \ell 10)]]{\subseteq}\mathsf{h}_i$$
$$[\ell 5{\mapsto}[f_1{\mapsto}(\ell_5, \ell 12), f_2{\mapsto}(\ell 7, \ell 8)]]{\subseteq}\mathsf{h}_j$$

And $\sigma_i = (\mathsf{s}_i, \mathsf{h}_i)$ and $\sigma_j = (\mathsf{s}_j, \mathsf{h}_j)$. $\mathsf{MergeStates}(\{\sigma_i, \sigma_j\}) = (\mathsf{s}, \mathsf{h})$, where

Algorithm 10 $\mathsf{MergeStates} \stackrel{\mathsf{def}}{=} \mathsf{StateSet} \to \mathsf{State}$

```
 1: procedure MergeStates(states)
 2:     merged_s ← fresh Store
 3:     for each σ ∈ states do
 4:         Dom(merged_s) ← Dom(merged_s) ∪ Dom(σ.s)
 5:     end for
 6:     for each v ∈ Dom(merged_s) do
 7:         for each σ ∈ states do
 8:             if v ∈ Dom(σ.s) then
 9:                 merged_s ← merged_s[v↦σ.s(v)]
10:             end if
11:         end for
12:     end for
13:     merged_h ← fresh Heap
14:     for each σ ∈ states do
15:         Dom(merged_h) ← Dom(merged_h) ∪ Dom(σ.h)
16:     end for
17:     for each ℓ_base ∈ Dom(merged_h) do
18:         for each σ ∈ states do
19:             if ℓ_base ∈ Dom(σ.h) then
20:                 if merged_h(ℓ_base) = () then
21:                     merged_h ← merged_h[ℓ_base↦σ.h(ℓ_base)]
22:                 else if ∃f ∈ Dom(obj')· then
23:                     [ℓ_base↦obj']⊆σ.h ∧ [ℓ_base↦obj]⊆merged_h ∧ obj'(f) ≠ obj(f)
24:                     merged_h ← merged_h[ℓ_base↦obj[f↦obj'(f)]]
25:                 end if
26:             end if
27:         end for
28:     end for
29:     return (merged_s, merged_h)
30: end procedure
```

$$[v{\mapsto}(\ell_1,\ell 2), x{\mapsto}(\ell 3,\ell 4), y{\mapsto}(\ell 5,\ell 6)]{\subseteq}\mathsf{s}$$

$$[\ell 5{\mapsto}[f_1{\mapsto}(\ell_5,\ell 12), f_2{\mapsto}(\ell 7,\ell 8)], \ell 9{\mapsto}[f_1{\mapsto}(\ell 9,\ell 10)]]{\subseteq}\mathsf{h} \quad \square$$

A.3 Algorithm Definitions for Static Execution Rules

A.3.1 Equal

TID and Issuer are integers so the usual equality rules apply. The special case for Issuer is $\perp$, in which case we define $\perp = \perp$.

$\mathsf{Equal} \stackrel{\mathsf{def}}{=} \mathsf{Scale} \times \mathsf{Scale} \to \mathsf{Bool}$:

$$\mathsf{Equal}(\epsilon, \epsilon) = \mathsf{True} \qquad \mathsf{Equal}(1, \epsilon)^{\dagger} = \mathsf{False} \qquad \mathsf{Equal}(1, 1) = \mathsf{True}$$

$\mathsf{Equal} \stackrel{\mathsf{def}}{=} \mathsf{Coord} \times \mathsf{Coord} \to \mathsf{Bool}$:

$$\mathsf{Equal}(\perp, \perp) = \mathsf{True} \qquad \mathsf{Equal}(\mathcal{A}, \mathcal{A}) = \mathsf{True} \qquad \mathsf{Equal}(\mathcal{A}, \perp)^{\dagger} = \mathsf{False}$$

$$\mathsf{Equal}(\mathcal{L}, \perp)^{\dagger} = \mathsf{False} \qquad \mathsf{Equal}(\mathcal{L}, \mathcal{A})^{\dagger} = \mathsf{False}$$

$$\mathsf{Equal}(\mathcal{L}(\ell_1), \mathcal{L}(\ell_2)) = \mathsf{True} \text{ if } \ell_1 = \ell_2 \quad \mathsf{Equal}(\mathcal{L}(\ell_1), \mathcal{L}(\ell_2)) = \mathsf{False} \text{ if } \ell_1 \neq \ell_2$$

† Equality is symmetric. $scale_1 = scale_2 \stackrel{\mathsf{def}}{=} \mathsf{Equal}(scale_1, scale_2)$; $coord_1 = coord_2 \stackrel{\mathsf{def}}{=} \mathsf{Equal}(coord_1, coord_2)$.

A.3.2 IsMemberOfARSet

$\mathsf{IsMemberOfARSet} \stackrel{\mathsf{def}}{=} \mathsf{AR} \times \mathsf{ARSet} \to \mathsf{Bool}$ given in Algorithm 11 is a membership predicate over an AR and ARSet. $ar \in ars \stackrel{\mathsf{def}}{=} \mathsf{IsMemberOfARSet}(ar, ars)$.

Example A.16 (IsMemberOfARSet)**.**

$\mathsf{IsMemberOfARSet}((1, \epsilon, \perp, \perp), \{(1, \epsilon, \perp, \perp)\}) = \mathsf{True}$.

Algorithm 11 $\mathsf{IsMemberOfARSet} \stackrel{\mathsf{def}}{=} \mathsf{AR} \times \mathsf{ARSet} \to \mathsf{Bool}$

```
1: procedure IsMemberOfARSet(ar, ars)
2:     for each ar' ∈ ars do
3:         if ar'.TID = ar.TID ∧ ar'.Scale ≥ ar.Scale ∧
   ar'.Coord = ar.Coord ∧
   ar'.Issuer = ar.Issuer then
4:             return True
5:         end if
6:     end for
7:     return False
8: end procedure
```

$\mathsf{IsMemberOfARSet}((1, \epsilon, \bot, \bot), \{(1, 1, \bot, \bot)\}) = \mathsf{True}.$

$\mathsf{IsMemberOfARSet}((1, 1, \bot, \bot), \{(1, 1, \bot, \bot)\}) = \mathsf{True}.$

$\mathsf{IsMemberOfARSet}((1, 1, \bot, \bot), \{(1, \epsilon, \bot, \bot)\}) = \mathsf{False}.$ □

A.3.3 $\mathsf{Add}_{\mathsf{AR}}$

$\mathsf{Add}_{\mathsf{AR}} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Scale} \times \mathsf{LocationSet} \to \mathsf{AM}$ given in Algorithm 12 adds a new AR to an AM with the specified scale for the memory locations provided. The returned AM differs to E.AM by containing a new AR for each of the memory locations given. We informally state the restriction that access requirements may only be added to a AM via $\mathsf{Add}_{\mathsf{AR}}$.

Example A.17 ($\mathsf{Add}_{\mathsf{AR}}$)**.** Let env be an environment E such that:

$$env.\mathsf{TID}{=}1 \qquad env.\mathsf{Coord}{=}\bot \qquad env.\mathsf{Issuer}{=}\bot$$

$$[\ell 1 \mapsto \{(1, \epsilon, \bot, \bot)\}] \subseteq env.\mathsf{AM}$$

$\mathsf{Add}_{\mathsf{AR}}(env, \epsilon, \{\ell 1, \ell 2\}) = am$, where $[\ell 1 \mapsto \{(1, \epsilon, \bot, \bot)$ $(1, \epsilon, \bot, \bot)$ $\}$, $\ell 2 \mapsto \{(1, \epsilon, \bot, \bot)\}]$ $\subseteq am$. There are two things to note here: (i) the read AR we

Algorithm 12 $\mathsf{Add_{AR}} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Scale} \times \mathsf{LocationSet} \to \mathsf{AM}$

```
1: procedure Add_AR(env, scale, locs)
2:     am ← env.AM
3:     for each ℓ ∈ locs do
4:         if ∃ar ∈ am(ℓ) ·
   ar.TID=env.TID ∧ ar.Scale < scale ∧ ar.Coord=env.Coord ∧
   ar.Issuer=env.Issuer then
5:             am(ℓ) ← am(ℓ)\{ar} ▷ Eliminate read AR; write AR subsumes it.
6:         end if
7:         am(ℓ) ← am(ℓ) ∪ {(env.TID, scale, env.Coord, env.Issuer)}
8:     end for
9:     return am
10: end procedure
```

are attempting to add to $env.\mathsf{AM}(\ell 1)$ already exists, so it is not added, highlighted in red; and (ii) $\ell 2$ did not exist in the domain of $env.\mathsf{AM}$ before the application of $\mathsf{Add_{AR}}$, so $\ell 2$ is added to the domain of am and associated with a read AR, highlighted in yellow. In (ii) we assert that issuing a AR on a memory location ℓ not in the domain of an access mapping am has the effect of adding ℓ to the domain of the returned access mapping am. □

Example A.18 ($\mathsf{Add_{AR}}$: Write AR elimination of read AR.)**.** Let env be an environment E such that:

$$env.\mathsf{TID}{=}1 \qquad env.\mathsf{Coord}{=}\bot \qquad env.\mathsf{Issuer}{=}\bot$$

$$[\ell 1{\mapsto}\{(1,\epsilon,\bot,\bot)\},\ell 2{\mapsto}\{(1,\epsilon,\bot,\bot)\}]{\subseteq}env.\mathsf{AM}$$

$\mathsf{Add_{AR}}(\mathsf{E},1,\{\ell 1,\ell 2\}) = am$, where $[\ell 1{\mapsto}\{(1,\epsilon,\bot,\bot)\ (1,1,\bot,\bot)\},$ $\ell 2{\mapsto}\{(1,\epsilon,\bot,\bot)\ (1,1,\bot,\bot)\}]{\subseteq}am$. The write access requirements, highlighted in yellow, issued to $\ell 1$ and $\ell 2$ subsume the existing read access requirements, highlighted in red, on $\ell 1$ and $\ell 2$ as the write and read access requirements differ

only in their scale, and $1 > \epsilon$. This models the semantics of permissions given in Boyland [2003]. That is, if one has write permission then one has read and write permission. Note however that we extend this concept to to be context aware w.r.t. the thread, coordination type and coordination instance. □

A.3.4 MergeAMs

$\mathsf{MergeAMs} \stackrel{\mathsf{def}}{=} \mathsf{AM} \times \mathsf{AM} \to \mathsf{AM}$ in Algorithm 13 takes two access mappings and returns an access mapping whose domain and co-domain is the union of the domain and co-domain of the access mappings given as arguments.

Algorithm 13 $\mathsf{MergeAMs} \stackrel{\mathsf{def}}{=} \mathsf{AM} \times \mathsf{AM} \to \mathsf{AM}$

```
1: procedure MergeAMs(am_1, am_2)
2:     merged ← fresh AM
3:     Dom(merged) ← Dom(am_1) ∪ Dom(am_2)
4:     for each ℓ ∈ Dom(merged) do
5:         if ℓ ∈ Dom(am_1) ∧ ℓ ∉ Dom(am_2) then
6:             merged ← merged[ℓ↦am_1(ℓ)]
7:         else if ℓ ∈ Dom(am_2) ∧ ℓ ∉ Dom(am_1) then
8:             merged ← merged[ℓ↦am_2(ℓ)]
9:         else
10:            merged ← merged[ℓ↦am_1(ℓ) ∪ am_2(ℓ)]
11:        end if
12:    end for
13:    return merged
14: end procedure
```

Example A.19 (MergeAMs)**.** Let am_1 and am_2 be access mappings AM such that:

$$[\ell 1 \mapsto \{(1, \epsilon, \mathcal{L}(\ell 2), 3), (2, 1, \mathcal{A}, 2)\}, \ell 2 \mapsto \{(1, \epsilon, \bot, \bot)\}] \subseteq am_1$$

$$[\ell 1 \mapsto \{(2, \epsilon, \bot, \bot), (1, \epsilon, \mathcal{L}(\ell 2), 3)\}] \subseteq am_2$$

$\mathsf{MergeAMs}(am_1, am_2) = merged$, where

$[\ell 1 \mapsto \{(1, \epsilon, \mathcal{L}(\ell 2), 3), (2, 1, \mathcal{A}, 2), (2, \epsilon, \bot, \bot)\}, \ell 2 \mapsto \{(1, \epsilon, \bot, \bot)\}] \subseteq merged.$ $\square$

A.3.5 CreateObject

$\mathsf{CreateObject} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Type} \to \mathsf{Object} \times \mathsf{LocationSet}$ given in Algorithm 14 creates an object for a type cn. The first component of the returned tuple is an object $[f_1 \mapsto (\ell_1, \mathsf{null}), \ldots, f_n \mapsto (\ell_n, \mathsf{null})]$ where $\{f_1, \ldots, f_n\} \subseteq \mathsf{TypeFields}(cn)$; the second component is the set of memory locations $\{\ell_1, \ldots, \ell_n\}$ associated with the fields of the object. $\mathsf{TypeFields} \stackrel{\mathsf{def}}{=} \mathsf{Type} \to \mathsf{FieldSet}$ returns the set of fields a type comprises. We assume this information is derivable from the program text. We carry the environment E to give context to $\mathsf{fresh}\ \ell$.

Algorithm 14 $\mathsf{CreateObject} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Type} \to \mathsf{Object} \times \mathsf{LocationSet}$

```
1:  procedure CreateObject(env, cn)
2:      obj ← fresh Object
3:      locs ← {}
4:      Dom(obj) ← TypeFields(cn)
5:      for each f ∈ Dom(obj) do
6:          ℓ_f ← fresh ℓ
7:          locs ← locs ∪ {ℓ_f}
8:          obj ← obj[f ↦ (ℓ_f, null)]
9:      end for
10:     return (obj, locs)
11: end procedure
```

Example A.20 ($\mathsf{CreateObject}$)**.** Let env be an environment E such that $env.\mathsf{FS} = \{\ell 1, \ell 2\}$. Further, assume that `Node` is a valid instance of Type.

$\mathsf{CreateObject}(env, \texttt{Node}) = (obj, locs)$, where $[\texttt{next} \mapsto (\ell 3, \mathsf{null}), \texttt{value} \mapsto (\ell 4, \mathsf{null})] \subseteq obj$ and $locs = \{\ell 3, \ell 4\}$. $\square$

A.3.6 BaseLoc

$\mathsf{BaseLoc} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \rightarrow \mathsf{Location}$ given in Algorithm 15 returns the base memory location of an object a variable refers to. $\mathsf{CoDom}(M)$ returns the co-domain of a mapping M, $\mathsf{snd}((a, b)) = b$ and $\mathsf{Head}(\{f_1, \ldots, f_n\}) = f_1$.

Algorithm 15 $\mathsf{BaseLoc} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \rightarrow \mathsf{Location}$

1: **procedure** $\mathsf{BaseLoc}(env,\ v)$
2: $\quad \ell \leftarrow \mathsf{snd}(env.\mathsf{Var}(v))$
3: $\quad \exists obj \in \mathsf{CoDom}(env.\mathsf{Obj})\ \cdot \ell \in \mathsf{FieldLocations}(obj) \wedge$
4: $\quad\quad \ell_{\text{base}} = \mathsf{Head}(\mathsf{FieldLocations}(obj))$
5: $\quad$ **return** ℓ_{base}
6: **end procedure**

Example A.21 (BaseLoc)**.** Let env be an environment E such that:

$$[x \mapsto (\ell 1, \ell 3)] \subseteq env.\mathsf{Var} \quad [\ell 2 \mapsto [\texttt{next} \mapsto (\ell 2, \mathsf{null}), \texttt{value} \mapsto (\ell 3, \mathsf{null})]] \subseteq env.\mathsf{Obj}$$

$$\mathsf{BaseLoc}(env, x) = \ell 2. \quad \square$$

A.3.7 FieldLocations

$\mathsf{FieldLocations} \stackrel{\mathsf{def}}{=} \mathsf{Object} \rightarrow \mathsf{LocationSet}$ returns the set of memory locations associated with the fields of an object.

Example A.22 (FieldLocations)**.** Let $[\texttt{f1} \mapsto (\ell 1, \ell 2), \texttt{f2} \mapsto (\ell 3, \ell 4)] \subseteq obj$.

$\mathsf{FieldLocations}(obj) = \{\ell 1, \ell 3\}$. $\square$

A.3.8 FldLoc

$\mathsf{FldLoc} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \times \mathsf{Field} \rightarrow \mathsf{Location}$ given in Algorithm 16 returns the memory location associated with a field in an indirection.

Algorithm 16 $\mathsf{FldLoc} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \times \mathsf{Field} \rightarrow \mathsf{Location}$

```
1: procedure FldLoc(env, v, f)
2:     ℓ_base ← BaseLoc(env, v)
3:     obj ← env.Obj(ℓ_base)
4:     ∃f′ ∈ ObjectFields(obj) · f′=f ∧
5:         [..., f′↦(ℓ, val), ...]⊆obj
6:     return ℓ
7: end procedure
```

Example A.23 (FldLoc)**.** Let env be an evironment E such that:

$$[x{\mapsto}(\ell 1, \ell 2)]{\subseteq}env.\mathsf{Var} \quad [\ell 2{\mapsto}[\texttt{next}{\mapsto}(\ell 2, \mathsf{null}), \texttt{value}{\mapsto}(\ell 3, \mathsf{null})]]{\subseteq}env.\mathsf{Obj}$$

$$\mathsf{FldLoc}(env, x, \texttt{next}) = \ell 2. \quad \square$$

A.3.9 ObjectFields

$\mathsf{ObjectFields} \stackrel{\mathsf{def}}{=} \mathsf{Object} \rightarrow \mathsf{FieldSet}$ returns the fields an object entails.

Example A.24 (ObjectFields)**.** Let $[\texttt{f1}{\mapsto}(\ell 1, \ell 2), \texttt{f2}{\mapsto}(\ell 3, \ell 4)]{\subseteq}obj$.

$\mathsf{ObjectFields}(obj) = \{\texttt{f1}, \texttt{f2}\}$. $\square$

A.3.10 FldUpd

$\mathsf{FldUpd} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \times \mathsf{Field} \times \mathsf{Location} \rightarrow \mathsf{Obj}$ given in Algorithm 17 returns an object mapping with the value of the specified field updated to the provided location.

Algorithm 17 $\mathsf{FldUpd} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \times \mathsf{Field} \times \mathsf{Location} \to \mathsf{Obj}$

```
1: procedure FldUpd(env, v, f, loc)
2:     ℓ_base ← BaseLoc(env, v)
3:     obj ← env.Obj(ℓ_base)
4:     ∃f′ ∈ ObjectFields(obj) · f′=f ∧
5:         [..., f′↦(ℓ, val), ...]⊆obj
6:     obj′ ← obj[f↦(ℓ, loc)]
7:     objMap ← env.Obj
8:     objMap ← objMap[ℓ_base↦obj′]
9:     return objMap
10: end procedure
```

Example A.25 (FldUpd)**.** Let env be an environment E such that:

$$[x\mapsto(\ell 1, \ell 2)]\subseteq env.\mathsf{Var} \quad [\ell 2\mapsto[\texttt{next}\mapsto(\ell 2, \mathsf{null}), \texttt{value}\mapsto(\ell 3, \mathsf{null})]]\subseteq env.\mathsf{Obj}$$

$\mathsf{FldUpd}(env, x, \texttt{next}, \ell 4)=obj'$ where $[\ell 2\mapsto[\texttt{next}\mapsto(\ell 2, \ell 4), \texttt{value}\mapsto(\ell 3, \mathsf{null})]]\subseteq obj'$.

□

A.3.11 FldVal

$\mathsf{FldVal} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \times \mathsf{Field} \to \mathsf{Location}$ given in Algorithm 18 returns the value of an object's field.

Algorithm 18 $\mathsf{FldVal} \stackrel{\mathsf{def}}{=} \mathsf{E} \times \mathsf{Variable} \times \mathsf{Field} \to \mathsf{Location}$

```
1: procedure FldLoc(env, v, f)
2:     ℓ_base ← BaseLoc(env, v)
3:     obj ← env.Obj(ℓ_base)
4:     ∃f′ ∈ ObjectFields(obj) · f′=f ∧
5:         [..., f′↦(ℓ, val), ...]⊆obj
6:     return val
7: end procedure
```

Example A.26 (FldVal)**.** Let env be an environment E such that:

$$[x \mapsto (\ell 1, \ell 2)] \subseteq env.\mathsf{Var} \quad [\ell 2 \mapsto [\texttt{next} \mapsto (\ell 2, \ell 4), \texttt{value} \mapsto (\ell 3, \mathsf{null})]] \subseteq env.\mathsf{Obj}$$

$\mathsf{FldVal}(env, x, \texttt{next}) = \ell 4.$ □

A.3.12 Receiver

$\mathsf{Receiver} \stackrel{\mathsf{def}}{=} \mathsf{DeferredMethodCall} \to \mathsf{Variable}$ takes a deferred method call and gives you back the receiver of the method call.

Example A.27 (Receiver)**.** $\mathsf{Receiver}(\texttt{l.add(1)@ctxt}) = \texttt{l}.$ □

A.3.13 CollectReceivers

$\mathsf{CollectReceivers} \stackrel{\mathsf{def}}{=} \mathsf{DeferredMethodCallList} \to \mathsf{VariableSet}$ given in Algorithm 19 takes a list of deferred method calls and returns the set of receiver variables for those method calls.

Example A.28 (CollectReceivers)**.** $\mathsf{CollectReceivers}([\texttt{l.add(1)@ctxt}, \texttt{n.traverse()@ctxt}]) = \{\texttt{l}, \texttt{n}\}.$ □

Algorithm 19 $\mathsf{CollectReceivers} \stackrel{\mathsf{def}}{=} \mathsf{DeferredMethodCallList} \to \mathsf{VariableSet}$

```
1: procedure CollectReceivers(methodCalls)
2:     receivers ← {}
3:     for each methodCall ∈ methodCalls do
4:         receivers ← receivers ∪ {Receiver(methodCall)}
5:     end for
6:     return receivers
7: end procedure
```

A.3.14 ReceiverCalls

ReceiverCalls $\stackrel{\text{def}}{=}$ DeferredMethodCallList $\times$ Variable $\rightarrow$ DeferredMethodCallList given in Algorithm 20 takes a list of method calls and a receiver variable and returns the list of deferred method calls issued on the given receiver variable. ReceiverCalls preserves program order, $\xrightarrow{po}$.

Example A.29 (ReceiverCalls)**.** ReceiverCalls([`l.add(1)`@ctxt, `n.traverse()`@ctxt, `l.traverse()`@ctxt], l) = [`l.add(1)`@ctxt, `l.traverse`@ctxt]. □

Algorithm 20 ReceiverCalls $\stackrel{\text{def}}{=}$ DeferredMethodCallList $\times$ Variable $\rightarrow$ DeferredMethodCallList

```
1: procedure ReceiverCalls(methodCalls, v)
2:     callsOnReceiver ← [ ]
3:     for each methodCall ∈ methodCalls do
4:         if Receiver(methodCall) = v then
5:             callsOnReceiver ← methodCall :: callsOnReceiver
6:         end if
7:     end for
8:     return callsOnReceiver
9: end procedure
```

A.3.15 Sort

Sort $\stackrel{\text{def}}{=}$ DeferredMethodCallList $\times$ Comparator $\rightarrow$ DeferredMethodCallList takes a list of deferred method calls and a comparator and returns an ordered list of deferred method calls using the provided comparator. We assume that Sort is *stable*. That is, if $a < b$, and there exists instances of both a and b, a_i and a_j and b_i and b_j, such that in the list to be sorted $[a_i, b_i, a_j, b_j]$, then in the sorted list a_i will appear before a_j and b_i before b_j, e.g. $[a_i, a_j, b_i, b_j]$.

Example A.30 (Sort)**.** Given the list of method calls *methodCalls*=[`l.traverse()`@ctxt, `l.add(1)`@ctxt, `l.traverse()`@ctxt], where `l` is of type `LinkedList`, Sort(*methodCalls*, `LinkedList`@serialise) = [`l.add(1)`@ctxt, `l.traverse()`@ctxt, `l.traverse()`@ctxt]. □

A.3.16 ListToCmdSeq

ListToCmdSeq $\stackrel{\text{def}}{=}$ DeferredMethodCallList $\rightarrow$ DeferredMethodCallSequence returns a sequence of deferred method calls that preserves the ordering of the deferred method calls in the given deferred method call list.

Example A.31 (ListToCmdSeq)**.** ListToCmdSeq([`l.add(1)`@ctxt, `l.traverse()`@ctxt]) = `l.add(1)`@ctxt;`l.traverse()`@ctxt; □

A.3.17 Serialise

Serialise $\stackrel{\text{def}}{=}$ DeferredMethodCallList $\rightarrow$ DeferredMethodCallSequence given in Algorithm 21 takes a list of deferred method calls and serialises those method calls for each receiver according to its defining class's @serialise annotation. $list_1 ++ list_2 = list_3$, where $list_3$ contains the elements of $list_1$ and $list_2$ where $list_1, list_2$ and $list_3$ are instances of DeferredMethodCallList.

Example A.32 (Serialise)**.** Serialise([`l.traverse()`@ctxt, `n.add(1)`@ctxt, `l.add(2)`@ctxt, `n.traverse()`@ctxt]) = `l.add(2)`@ctxt; `l.traverse()`@ctxt; `n.add(1)`@ctxt; `n.traverse()`@ctxt; □

Algorithm 21 Serialise $\stackrel{\text{def}}{=}$ DeferredMethodCallList $\rightarrow$ DeferredMethodCallSequence

```
1: procedure Serialise(methodCalls)
2:     receivers ← CollectReceivers(methodCalls)
3:     serialised ← [ ]
4:     for each receiver ∈ receivers do
5:         receiverCalls ← ReceiverCalls(methodCalls, receiver)
6:         serialised ← Sort(receiverCalls, TypeOf(receiver)@serialise)
7:             ++ serialised
8:     end for
9:     return ListToCmdSeq(serialised)
10: end procedure
```

A.3.18 CheckSafeIO

CheckSafeIO $\stackrel{\text{def}}{=}$ E $\rightarrow$ Bool given in Algorithm 22 is a predicate that asserts the environment's coordination type is strong enough to perform an irreversible operation, e.g. `print`. The semantics of CheckSafeIO models that of a weakly isolated STM Harris et al. [2010]. CheckSafeIO does not prohibit use of the privitisation/publication idioms Spear et al. [2007].

Algorithm 22 CheckSafeIO $\stackrel{\text{def}}{=}$ E $\rightarrow$ Bool

```
1: procedure CheckSafeIO(env)
2:     if env.Coord = A then
3:         return False                    ▷ Transaction could abort.
4:     else
5:         return True
6:     end if
7: end procedure
```

Example A.33 (CheckSafeIO)**.** Let env be an environment E such that env.Coord=$\mathcal{A}$. CheckSafeIO(env)=False. □

A.4 Algorithm Definitions for Isolated?

A.4.1 Writes

Writes $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ ARSet given in Algorithm 23 filters the write access requirements from the set of access requirements specified.

Algorithm 23 Writes $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ ARSet

```
1: procedure Writes(prs)
2:     write_prs ← {}
3:     for each pr ∈ prs do
4:         if pr.Scale = 1 then
5:             write_prs ← write_prs ∪ {pr}
6:         end if
7:     end for
8:     return write_prs
9: end procedure
```

Example A.34 (Writes). Let $ars = \{(1, \epsilon, \bot, \bot), (2, 1, \bot, \bot), (3, 1, \bot, \bot)\}$. Writes$(ars) = \{(2, 1, \bot, \bot), (3, 1, \bot, \bot)\}$. □

A.4.2 AccessingTIDs

AccessingTIDs $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ Int given in Algorithm 24 returns the number of distinct threads that issue access requirements in the specified set of access requirements. $|s|$ is the cardinality of the set s.

Example A.35 (AccessingTIDs). Let $ars = \{(1, \epsilon, \bot, \bot), (2, 1, \bot, \bot), (3, 1, \bot, \bot), (1, 1, \mathcal{A}, 3)\}$. AccessingTIDs$(ars) = 3$. □

Algorithm 24 AccessingTIDs $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ Int

```
1: procedure AccessingTIDs(ars)
2:     tids ← {}
3:     for each ar ∈ ars do
4:         tids ← tids ∪ {ar.TID}
5:     end for
6:     return |tids|
7: end procedure
```

A.4.3 NumberOfWritingThreads

NumberOfWritingThreads $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ Int given in Algorithm 25 returns the number of distinct threads that issue write access requirements in the set of access requirements specified.

Algorithm 25 NumberOfWritingThreads $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ Int

```
procedure NumberOfWritingThreads(ars)
    return AccessingTIDs(Writes(ars))
end procedure
```

Example A.36 (NumberOfWritingThreads)**.** Let $ars = \{(1, \epsilon, \bot, \bot), (2, 1, \bot, \bot), (3, 1, \bot, \bot), (1, 1, \mathcal{A}, 3)\}$. NumberOfWritingThreads$(ars) = 3$. □

A.4.4 Reads

Reads $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ ARSet given in Algorithm 26 filters the read access requirements from the set of access requirements specified.

Example A.37 (Reads)**.** Let $ars = \{(1, \epsilon, \bot, \bot), (2, 1, \bot, \bot), (3, 1, \bot, \bot)\}$. Reads$(ars) = \{(1, \epsilon, \bot, \bot)\}$. □

Algorithm 26 Reads $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ ARSet

```
1: procedure Reads(ars)
2:     read_ars ← {}
3:     for each ar ∈ ars do
4:         if ar.Scale = ϵ then
5:             read_ars ← read_ars ∪ {pr}
6:         end if
7:     end for
8:     return read_ars
9: end procedure
```

A.4.5 RemoveReadsByTID

RemoveReadsByTID $\stackrel{\text{def}}{=}$ ARSet $\times$ TID $\rightarrow$ ARSet given in Algorithm 27 returns a set of access requirements minus the access requirements issued by the specified thread.

Algorithm 27 RemoveReadsByTID $\stackrel{\text{def}}{=}$ ARSet $\times$ TID $\rightarrow$ ARSet

```
1: procedure RemoveReadsByTID(ars, tid)
2:     filtered ← {}
3:     for each ar ∈ ars do
4:         if ar.TID ≠ tid then
5:             filtered ← filtered ∪ {ar}
6:         end if
7:     end for
8:     return filtered
9: end procedure
```

Example A.38 (RemoveReadsByTID)**.** Let $ars = \{(1, \epsilon, \bot, \bot), (2, \epsilon, \bot, \bot), (1, \epsilon, \mathcal{A}, 3)\}$. RemoveReadsByTID$(ars, 1) = \{(2, \epsilon, \bot, \bot)\}$. $\square$

A.4.6 PartitionAccessesByCoordType

PartitionAccessesByCoordType $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ ARSet $\times$ ARSet $\times$ ARSet given in Algorithm 28 partitions a set of access requirements into a triple of access requirement sets. The first component of the returned triple comprises the access requirements issued under no coordination semantics; the second those issued transactionally; and the third those issued by locks. The syntax $partitioned[i]$ where $0 \leq i < 3$ accesses the ith component of the triple $partitioned$.

Algorithm 28 PartitionAccessesByCoordType $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ ARSet $\times$ ARSet $\times$ ARSet

```
1: procedure PartitionAccessesByCoordType(ars)
2:     partitioned ← ({}, {}, {})
3:     component_index ← 0
4:     coord_types ← {⊥, A, L}
5:     for each coord ∈ coord_types do
6:         for each ar ∈ ars do
7:             if ar.Coord = coord then
8:                 partitioned[component_index] ←
   partitioned[component_index] ∪ {ar}
9:             end if
10:        end for
11:        component_index ← component_index + 1
12:    end for
13:    return partitioned
14: end procedure
```

Example A.39 (PartitionAccessesByCoordType)**.** Let $ars = \{(1, 1, \bot, \bot), (1, 1, \mathcal{A}, 3), (2, 1, \mathcal{L}(p2), 4), (1, \epsilon, \mathcal{A}, 5)\}$. PartitionAccessesByCoordType$(ars) = (\{(1, 1, \bot, \bot)\}, \{(1, 1, \mathcal{A}, 3), (1, \epsilon, \mathcal{A}, 5)\}, \{(2, 1, \mathcal{L}(p2), 4)\})$. $\square$

A.4.7 TransactionsAccessMutex

TransactionsAccessMutex $\stackrel{\text{def}}{=}$ AM $\times$ AbsLocSet $\times$ ARSet $\rightarrow$ Bool given in Algorithm 29 asserts that *each* of the transactional instances in the set of transactionally issued access requirements accesses the memory location (the mutex) specified.

Algorithm 29 TransactionsAccessMutex $\stackrel{\text{def}}{=}$ AM $\times$ AbsLoc $\times$ ARSet $\rightarrow$ Bool

```
1: procedure TransactionsAccessMutex(am, mutex, ars)
2:     for each txn ∈ ars do
3:         if ∄ar ∈ am(mutex) · ar.Issuer = txn.Issuer then
4:             return False
5:         else
6:             goto 2
7:         end if
8:     end for
9:     return True
10: end procedure
```

Example A.40 (TransactionsAccessMutex)**.** Let am be an access mapping AM such that $[\ell 1 \mapsto \{(1, 1, \mathcal{A}, 3), (2, \epsilon, \perp, \perp), (4, \epsilon, \mathcal{A}, 4)\}] \subseteq am$, $mutex = \ell 1$ and $ars = \{\}$.

TransactionsAccessMutex$(am, mutex, ars)$ = True. The predicate trivially succeeds as $ars = \{\}$ results in the body of the **for each** being skipped. □

Example A.41 (TransactionsAccessMutex)**.** Let am be an access mapping AM such that $[\ell 1 \mapsto \{(1, 1, \mathcal{A}, 3), (2, \epsilon, \perp, \perp), (4, \epsilon, \mathcal{A}, 4)\}] \subseteq pm$, $mutex = \ell 1$ and $ars = \{(1, 1, \mathcal{A}, 3), (4, \epsilon, \mathcal{A}, 4)\}$. TransactionsAccessMutex$(am, mutex, ars)$ = True. The predicate succeeds as transactional instances 3 and 4 access $\ell 1$ in am. □

Example A.42 (TransactionsAccessMutex)**.** Let am be an access mapping AM such that $[\ell 1 \mapsto \{(1, 1, \mathcal{A}, 3), (2, \epsilon, \perp, \perp), (4, \epsilon, \mathcal{A}, 4)\}] \subseteq pm$, $mutex = \ell 1$ and

ars=$\{(1, 1, \mathcal{A}, 5), (4, \epsilon, \mathcal{A}, 4)\}$. TransactionsAccessMutex$(am, mutex, ars)$ = False. The predicate fails as transactional instance 5 does not access $\ell 1$ in am. □

A.4.8 LocksAgreeOnMutex

LocksAgreeOnMutex $\stackrel{\text{def}}{=}$ Location $\times$ ARSet $\rightarrow$ Bool given in Algorithm 30 asserts that all the specified lock access requirements are protected on the mutex provided.

Algorithm 30 LocksAgreeOnMutex $\stackrel{\text{def}}{=}$ AbsLoc $\times$ ARSet $\rightarrow$ Bool

```
1: procedure LocksAgreeOnMutex(ℓ, ars)
2:     if ars = {} then
3:         return True
4:     end if
5:     return ∄ar ∈ ars · ar.Coord ≠ ℓ
6: end procedure
```

Example A.43 (LocksAgreeOnMutex)**.** Let $ars = \{(1, 1, \mathcal{L}(\ell 1), 3), (2, \epsilon, \mathcal{L}(\ell 1), 4)\}$. LocksAgreeOnMutex$(\ell 1, ars)$ = True. The predicate succeeds as all lock permission requirements in ars uses the mutex $\ell 1$. □

Example A.44 (LocksAgreeOnMutex)**.** Let $ars = \{(1, 1, \mathcal{L}(\ell 1), 3), (2, \epsilon, \mathcal{L}(\ell 2), 4)\}$. LocksAgreeOnMutex$(\ell 1, ars)$ = False. The predicate fails as at least one lock permission requirement in ars uses a different mutex to that of $\ell 1$. □

A.4.9 FilterLocks

FilterLocks $\stackrel{\text{def}}{=}$ ARSet $\rightarrow$ ARSet given in Algorithm 31 filters the lock access requirements from the set of access requirements provided.

Algorithm 31 FilterLocks $\stackrel{\text{def}}{=}$ ARSet $\to$ ARSet

```
1: procedure FilterLocks(ars)
2:     lock_ars ← {}
3:     for each ar ∈ ars do
4:         if ar.Coord = L then
5:             lock_ars ← lock_ars ∪ {ar}
6:         end if
7:     end for
8:     return lock_ars
9: end procedure
```

Example A.45 (FilterLocks)**.** Let $ars = \{(1, 1, \mathcal{L}(\ell 1), 3), (2, \epsilon, \mathcal{A}, 2)\}$. FilterLocks$(ars) =$

$\{(1, 1, \mathcal{L}(\ell 1), 3)\}$. □

A.4.10 FilterTxns

FilterTxns $\stackrel{\text{def}}{=}$ ARSet $\to$ ARSet given in Algorithm 32 filters the transactional access requirements from the set of access requirements provided.

Algorithm 32 FilterTxns $\stackrel{\text{def}}{=}$ ARSet $\to$ ARSet

```
1: procedure FilterTxns(ars)
2:     txn_ars ← {}
3:     for each ar ∈ ars do
4:         if ar.Coord = A then
5:             txn_ars ← txn_ars ∪ {ar}
6:         end if
7:     end for
8:     return txn_ars
9: end procedure
```

Example A.46 (FilterTxns)**.** Let $ars = \{(1, 1, \mathcal{L}(\ell 1), 3), (2, \epsilon, \mathcal{A}, 2)\}$. FilterTxns$(ars) =$

$\{(2, \epsilon, \mathcal{A}, 2)\}$. □

A.4.11 LocksAndTxnsIsolated

LocksAndTxnsIsolated $\stackrel{\text{def}}{=}$ AM $\times$ ARSet $\times$ ARSet $\rightarrow$ Bool given in Algorithm 33 asserts that all lock and transactionally issued access requirements in the set of access requirements specified are isolated. LocksAndTxnsIsolated is an implementation of Definition 7.3 given in Section 10.2.2.

Algorithm 33 LocksAndTxnsIsolated $\stackrel{\text{def}}{=}$ AM $\times$ ARSet $\times$ ARSet $\rightarrow$ Bool

```
1:  procedure LocksAndTxnsIsolated(am,lk,txn)
2:      for each lk_access ∈ lk do
3:          remaining_accesses ← RemoveAccessesByTID(lk∪txn, lk_access.TID)
4:          if lk_access.Scale = ϵ then
5:              remaining_accesses ← remaining_accesses\Reads(remaining_accesses)
6:          end if
7:          mutex_used ← lk_access.Coord
8:          if LocksAgreeOnMutex(mutex_used, FilterLocks(remaining_accesses)) ∧
    TransactionsAccessMutex(am, mutex_used, FilterTxns(remaining_accesses))
    then
9:              goto 2
10:         else
11:             return False
12:         end if
13:     end for
14:     return True
15: end procedure
```

Example A.47 (LocksAndTxnsIsolated)**.** Consider the following program where only locks access `v` and `x`. We assert `v` resides at location $\ell 1$ and `x` at $\ell 2$.

```
Int v; Int x;
```

```
v := 0; x := 0;
```

```
1:sync(v) {  ||  3:sync(v) {
 v := 1;     ||   v := 2;
}            ||  }
2:sync(v){   ||  4:sync(v){
 x := v;     ||   x := v;
}            ||  }
```

Let am be the program's derived access mapping such that:

$$[\ell 1 \mapsto \{(1,1,\mathcal{L}(\ell 11),1),(1,\epsilon,\mathcal{L}(\ell 1),2),(2,1,\mathcal{L}(\ell 1),3),(2,\epsilon,\mathcal{L}(\ell 1),4)\},$$
$$\ell 2 \mapsto \{(1,1,\mathcal{L}(\ell 1),2),(2,1,\mathcal{L}(\ell 1),4)\}] \subseteq am$$

Let $lk = \mathsf{FilterLocks}(am(\ell 1))$ and $txn = \mathsf{FilterTxns}(am(\ell 1))$:

$$lk = \{(1,1,\mathcal{L}(\ell 1),1),(1,\epsilon,\mathcal{L}(\ell 1),2),(2,1,\mathcal{L}(\ell 1),3),(2,\epsilon,\mathcal{L}(\ell 1),4)\} \quad txn = \{\}$$

$\mathsf{LocksAndTxnsIsolated}(am, lk, txn) = \mathsf{True}$. □

Example A.48 (LocksAndTxnsIsolated)**.** Consider the following program where locks and transactions access `v`, `x` and `y`. We assert `v` resides at location $\ell 1$, `x` at $\ell 2$ and `y` at $\ell 3$.

```
Int v; Int x; Int y;
```

```
v := 0; x := 0; y := 0;
```

```
1:sync(v) {  ||  3:atomic {
 v := 1;     ||   y := v;
}            ||  }
2:sync(x){   ||
 x := v;     ||
}            ||
```

Let am be the program's derived access mapping such that:

$$[\ell 1 \mapsto \{(1,1,\mathcal{L}(\ell 1),1),(1,\epsilon,\mathcal{L}(\ell 2),2),(2,\epsilon,\mathcal{A},3)\},$$
$$\ell 2 \mapsto \{(1,1,\mathcal{L}(\ell 2),2)\}, p3 \mapsto \{(2,1,\mathcal{A},3)\}] \subseteq am$$

Let $lk = \mathsf{FilterLocks}(am(\ell 1))$ and $txn = \mathsf{FilterTxns}(am(\ell 1))$:

$$lk = \{(1,1,\mathcal{L}(\ell),1),(1,\epsilon,\mathcal{L}(\ell 2),2)\} \quad txn = \{(2,\epsilon,\mathcal{A},3)\}$$

$\mathsf{LocksAndTxnsIsolated}(am, lk, txn) = \mathsf{True}$. □

Example A.49 (LocksAndTxnsIsolated)**.** Consider the following program where both locks and transactions write `v` which we assert resides at location $\ell 1$:

```
Int v;
v := 0;
```

```
1:atomic {  ||  3:atomic {
 v := 1;    ||   v := 3;
}           ||  }
2:sync(v){  ||  4:sync(v){
 v := 2;    ||   v := 4;
}           ||  }
```

Let am be the program's access mapping such that:

$$[\ell 1 \mapsto \{(1,1,\mathcal{A},1),(1,1,\mathcal{L}(\ell 1),2),(2,1,\mathcal{A},3),(2,1,\mathcal{L}(\ell 1),4)\}] \subseteq am$$

Let $lk = \mathsf{FilterLocks}(am(\ell 1))$ and $txn = \mathsf{FilterTxns}(am(\ell 1))$:

$$lk = \{(1,1,\mathcal{L}(\ell 1),2),(2,1,\mathcal{L}(\ell 1),4)\} \quad txn = \{(1,1,\mathcal{A},1),(2,1,\mathcal{A},3)\}$$

$\mathsf{LocksAndTxnsIsolated}(am, lk, txn) = \mathsf{True}$. □

Example A.50 (LocksAndTxnsIsolated)**.** Consider the following program where both locks and transactions write `v`. We assert `v` resides at location $\ell 1$ and `x` at $\ell 2$.

```
Int v; Int x;
v := 0; x := 0;
```

```
1:sync(v) {  ||  4:atomic {
 v := 1;     ||   v := 3;
}            ||  }
2:atomic{    ||
 x := v;     ||
}            ||
3:sync(x) }  ||
 v := 2;     ||
}            ||
```

Let am be the program's derived access mapping such that:

$$[\ell 1 \mapsto \{(1,1,\mathcal{L}(\ell 1),1),(1,\epsilon,\mathcal{A},2),(1,1,\mathcal{L}(\ell 2),3),(2,1,\mathcal{A},4)\},$$
$$\ell 2 \mapsto \{(1,1,\mathcal{A},2)\}] \subseteq am$$

Let $lk = \mathsf{FilterLocks}(am(\ell 1))$ and $txn = \mathsf{FilterTxns}(am(\ell 1))$:

$$lk = \{(1,1,\mathcal{L}(\ell 1),1),(1,1,\mathcal{L}(\ell 2),3)\} \quad txn = \{(1,\epsilon,\mathcal{A},2),(2,1,\mathcal{A},4)\}$$

$\mathsf{LocksAndTxnsIsolated}(am, lk, txn) = \mathsf{False}$. The predicate fails as transactional instance 4 does not access the mutex that lock instance 3's write of `v` is protected on. □

Example A.51 (LocksAndTxnsIsolated)**.** Consider the following program which is similar to that given in Example A.50.

```
Int v; Int x;
```

```
v := 0; x := 0;
```

```
1:sync(v) {  ||  4:atomic {
 v := 1;     ||   v := x;
}            ||  }
2:atomic{    ||
 x := v;     ||
}            ||
3:sync(x) }  ||
 v := 2;     ||
}            ||
```

Let am be the program's derived access mapping such that:

$$[\ell 1 \mapsto \{(1,1,\mathcal{L}(\ell 1),1),(1,\epsilon,\mathcal{A},2),(1,1,\mathcal{L}(\ell 2),3),(2,1,\mathcal{A},4)\},$$
$$\ell 2 \mapsto \{(1,1,\mathcal{A},2),(2,\epsilon,\mathcal{A},4)\}] \subseteq am$$

Let $lk = \mathsf{FilterLocks}(am(\ell 1))$ and $txn = \mathsf{FilterTxns}(am(\ell 1))$:

$$lk = \{(1,1,\mathcal{L}(\ell 1),1),(1,1,\mathcal{L}(\ell 2),3)\} \quad txn = \{(1,\epsilon,\mathcal{A},2),(2,1,\mathcal{A},4)\}$$

$\mathsf{LocksAndTxnsIsolated}(am, lk, txn) = \mathsf{True}$. The predicate succeeds as transactional instance 4 accesses the mutexes used by lock instances 1 and 3. □

Appendix B

Example Applications of Part II's Static Framework

In all examples we assert am is an instance of an access mapping AM. Each memory location $\ell \in \mathsf{Dom}(am)$ is annotated with a label to aid in presentation. For example, $v\,\ell 1$ in the presentation of am denotes that $\ell 1$ is the memory location that represents the location of the variable v. The names of memory locations in the examples can be derived by $\mathsf{fresh}\ \ell$ yielding a memory location with a strictly increasing integer label i, $\ell\ i$, where $i > 0$ and initially $i = 1$. For example, given `Node n1; Node n2;`, the first application of ($\underline{\mathsf{VAR{-}DECL}}$) sees `n1` being associated with $\ell 1$ and the second application of ($\underline{\mathsf{VAR{-}DECL}}$) sees `n2` being associated with $\ell 2$. When describing rule applications we use the form $rule \times N$ to denote N successive applications of $rule$, e.g. $rule \times 2 = rule\ rule$. We use the syntax $rule\langle rule_1 \ldots rule_n \rangle$ to denote that the rules $rule_1 \ldots rule_n$ appear in the immediate derivation of $rule$. To keep the presentation of the examples concise we omit applications of the sequencing rules.

Example B.1 (Only Readers and Single Accessing Threads)**.** Consider the following program where v is read by threads 1 and 2, thread 1 writes x and thread 2 writes y and z.

Program.

```
Int v; Int x; Int y; Int z;
v := 0; x := 0; y := 0; z := 0;
```

Thread 1	**Thread 2**
`x := v;`	`y := v;`
	`z := v;`

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL) × 4, (ASSIGN−INT−LITERAL) × 4;
  - Thread 1: (ASSIGN−VAR−LITERAL);
  - Thread 2: (ASSIGN−VAR−LITERAL) × 2.
- ⟩

Access Mapping.

$$
\begin{aligned}
&[\, v\ \ell 1 \mapsto \{(1, \epsilon, \bot, \bot), (2, \epsilon, \bot, \bot)\},\\
&\ x\ \ell 2 \mapsto \{(1, 1, \bot, \bot)\},\\
&\ y\ \ell 3 \mapsto \{(2, 1, \bot, \bot)\},\\
&\ z\ \ell 4 \mapsto \{(2, 1, \bot, \bot)\}] \subseteq am
\end{aligned}
$$

Isolation.

Isolated?(am) = True. **C1** applies for v as $\ell 1$ is only read; **C1** applies for x, y and z as $\ell 2, \ell 3$ and $\ell 4$ are all accessed by a single thread. □

Example B.2 (Several Accessing Threads; Single Writer Thread; Uncoordinated Write)**.**

Program.

```
Int v; Int x;
v := 0; x := 0;
```

Thread 1	**Thread 2**
`v := 1;`	`x := v;`

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL) × 2, (ASSIGN−INT−LITERAL) × 2;
  - Thread 1: (ASSIGN−INT−LITERAL);
  - Thread 2: (ASSIGN−VAR−LITERAL).
- ⟩

Access Mapping.

$$[\, v\ \ell 1 \mapsto \{(1, 1, \bot, \bot), (2, \epsilon, \bot, \bot)\},$$
$$x\ \ell 2 \mapsto \{(2, 1, \bot, \bot)\}] \subseteq am$$

Isolation.

Isolated?(am) = False, due to **C2.1**. Thread 1's uncoordinated write of $\ell 1$ will not be isolated with the uncoordinated read of $\ell 1$ issued by thread 2. □

Example B.3 (Several Accessing Threads; Single Writer Thread; Uncoordinated Read)**.** We now present an example which triggers the second part of the disjunct of **C2.1**. That is, we have a single writing thread whose write is issued under a coordinated semantics, and an uncoordinated read issued to the same location outside of the writing thread.

Program.

```
Int v; Int x;
v := 0; x := 0;
```

Thread 1	**Thread 2**
1:atomic {	x := v;
v := 1;	
}	

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL) × 2, (ASSIGN−INT−LITERAL) × 2;
  - Thread 1: (TRANSACTION)⟨(ASSIGN−INT−LITERAL)⟩;
  - Thread 2: (ASSIGN−VAR−LITERAL).
- ⟩

Access Mapping.

$$[\,v\,\ell 1 \mapsto \{(1,1,\mathcal{A},1),(2,\epsilon,\perp,\perp)\},$$
$$x\,\ell 2 \mapsto \{(2,1,\perp,\perp)\}] \subseteq am$$

Isolation.

$\mathsf{Isolated?}(am) = \mathsf{False}$, due to **C2.1**. Thread 1's transactional write of $\ell 1$ will not be isolated with the uncoordinated read of $\ell 1$ issued by thread 2. □

Example B.4 (Several Accessing Threads; Single Writer Thread; Writer Thread's Writes Isolated w.r.t. Reads; Uncoordinated Read Issued by Writer Thread)**.** An uncoordinated read of a memory location ℓ can only exist in a program where several threads access ℓ if and only if: the uncoordinated read of ℓ is issued by the writing thread and the writes issued by the writing thread are isolated w.r.t. the reads of ℓ issued outside of the writing thread.

Program.

```
Int v; Int x; Int y;
v := 0; x := 0; y := 0;
```

Thread 1	**Thread 2**
`1:atomic {`	`2:atomic {`
`v := 1;`	`x := v;`
`}`	`}`
`y := v;`	

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL) × 3, (ASSIGN−INT−LITERAL) × 3;
  - Thread 1: (TRANSACTION)⟨(ASSIGN−INT−LITERAL)⟩, (ASSIGN−VAR−LITERAL);
  - Thread 2: (TRANSACTION)⟨(ASSIGN−VAR−LITERAL)⟩.
- ⟩

Access Mapping.

$$[\,v\,\ell 1{\mapsto}\{(1,1,\mathcal{A},1),(1,\epsilon,\perp,\perp),(2,\epsilon,\mathcal{A},2)\},$$
$$x\,\ell 2{\mapsto}\{(2,1,\mathcal{A},2)\},\ y\,\ell 3{\mapsto}\{(1,1,\perp,\perp)\}]{\subseteq}am$$

Isolation. $\mathsf{Isolated?}(am) = \mathsf{True}$. Due to **C2.2** thread 1's transactional write of $\ell 1$ is isolated with thread 2's transactional read of $\ell 1$. $\ell 2$ and $\ell 3$ are isolated due to **C1**. (See Section A.4.11 for examples of LocksAndTxnsIsolated.) □

Example B.5 (Several Accessing Threads; Single Writer Thread; Writer Thread's Writes not Isolated w.r.t. Reads; Uncoordinated Read Issued by Writer Thread)**.** We present a non-isolated version of Example B.4.

Program.

```
Int v; Int x; Int y;
v := 0; x := 0; y := 0;
```

Thread 1	**Thread 2**
`1:atomic {`	`2:sync(x) {`
`v := 1;`	`x := v;`
`}`	`}`
`y := v;`	

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL) × 3, (ASSIGN−INT−LITERAL) × 3;
  - Thread 1: (TRANSACTION)⟨(ASSIGN−INT−LITERAL)⟩, (ASSIGN−VAR−LITERAL);

– Thread 2: (LOCK)⟨(ASSIGN−VAR−LITERAL)⟩.

- ⟩

Access Mapping.

$$[\,v\,\ell 1 \mapsto \{(1,1,\mathcal{A},1),(1,\epsilon,\perp,\perp),(2,\epsilon,\mathcal{L}(\ell 2),2)\},$$
$$x\,\ell 2 \mapsto \{(2,1,\mathcal{L}(\ell 2),2)\},\; y\,\ell 3 \mapsto \{(1,1,\perp,\perp)\}] \subseteq am$$

Isolation.

Isolated?(am) = False, due to **C2.2**. Thread 1's transactional write of $\ell 1$ is not isolated with thread 2's lock issued read of $\ell 1$ as transactional instance 1 does not access lock instance 2's mutex, $\ell 2$. □

Example B.6 (Several Threads Issue Uncoordinated Writes)**.** If several threads issue uncoordinated writes to a memory location ℓ then all accesses to ℓ are subject to a data race.

Program.

```
Int v; Int x;
v := 0; x := 0;
```

Thread 1	**Thread 2**
`v := 1;`	`v := 2;`
`x := v;`	

Rule Applications.

- (PROGRAM)⟨

  – Main thread: (VAR−DECL) × 2, (ASSIGN−INT−LITERAL) × 2;

– Thread 1: (ASSIGN−INT−LITERAL), (ASSIGN−VAR−LITERAL);

– Thread 2: (ASSIGN−INT−LITERAL).

- ⟩

Access Mapping.

$$[\, v\ \ell 1 \mapsto \{(1,1,\bot,\bot),(1,\epsilon,\bot,\bot),(2,1,\bot,\bot)\},$$
$$x\ \ell 2 \mapsto \{(1,1,\bot,\bot)\}] \subseteq am$$

Isolation.

Isolated?(am) = False, due to **C2.3**. All accesses issued to $\ell 1$ are not isolated due to threads 1 and 2 issuing uncoordinated writes to $\ell 1$. □

Example B.7 (Only Transactional Accesses)**.** If all accesses to a memory location ℓ are issued tranasctionally then those accesses are trivially isolated.

Program.

```
Int v; Int x;
v := 0; x := 0;
```

Thread 1	**Thread 2**
`1:atomic {`	`2:atomic {`
`v := 1;`	`v := 2;`
`}`	`}`

Rule Applications.

- (PROGRAM)⟨

  – Main thread: (VAR−DECL) × 2, (ASSIGN−INT−LITERAL) × 2;

– Thread 1: (TRANSACTION)⟨(ASSIGN−INT−LITERAL)⟩;

– Thread 2: (TRANSACTION)⟨(ASSIGN−INT−LITERAL)⟩.

• ⟩

Access Mapping.

$$[\, v\, \ell 1 \mapsto \{(1,1,\mathcal{A},1),(2,1,\mathcal{A},2)\}] \subseteq am$$

Isolation.

Isolated?(am) = True. Due to **C3** all accesses issued to $\ell 1$ are isolated as threads 1 and 2 issue their writes of $\ell 1$ transactionally. □

Example B.8 (Only Lock Accesses)**.** Case **C4** covers two scenarios for accesses issued to a memory location ℓ: (1) all accesses to ℓ are issued by locks; and (2) accesses to ℓ are issued by locks and transactions. This example covers (1); subsequent examples cover (2).

Program.

```
Int v; Int x; Int y;
v := 0; x := 0; y := 0;
```

Thread 1	**Thread 2**
`1:sync(v) {`	`3:sync(v) {`
`  v := 1;`	`  y := v;`
`}`	`}`
`2:sync(x) }`	
`  x := v;`	
`}`	

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL) × 3, (ASSIGN−INT−LITERAL) × 3;
  - Thread 1: (LOCK)⟨(ASSIGN−INT−LITERAL)⟩,
    (LOCK)⟨(ASSIGN−VAR−LITERAL)⟩;
  - Thread 2: (LOCK)⟨(ASSIGN−VAR−LITERAL)⟩.
- ⟩

Access Mapping.

$$[\,v\;\ell 1 \mapsto \{(1,1,\mathcal{L}(\ell 1),1),(1,\epsilon,\mathcal{L}(\ell 2),2),(2,\epsilon,\mathcal{L}(\ell 1),3)\},$$
$$x\;\ell 2 \mapsto \{(1,1,\mathcal{L}(\ell 2),2)\},\; y\;\ell 3 \mapsto \{(2,1,\mathcal{L}(\ell 1),3)\}] \subseteq am$$

Isolation.

Isolated?(am) = True. Due to **C4** the accesses issued to $\ell 1$ by threads 1 and 2 are isolated: the write and read issued by thread 1 and respectively thread 2 are isolated as they both use the same mutex, $\ell 1$; lock instances 2 and 3 do not need to use the same mutex as both only read $\ell 1$. □

Example B.9 (Lock and Transactional Accesses)**.**

Program.

```
Int v; Int x; Int y;
v := 0; x := 0; y := 0;
```

Thread 1	Thread 2
1:sync(v) {	3:sync(v) {
v := 1;	y := v;
}	}
2:sync(x) {	4:atomic {
x := v;	y := v;
}	}

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL) × 3, (ASSIGN−INT−LITERAL) × 3;
  - Thread 1: (LOCK)⟨(ASSIGN−INT−LITERAL)⟩,
    (LOCK)⟨(ASSIGN−VAR−LITERAL)⟩;
  - Thread 2: (LOCK)⟨(ASSIGN−VAR−LITERAL)⟩,
    (TRANSACTION)⟨(ASSIGN−VAR−LITERAL)⟩.
- ⟩

Access Mapping.

$$[\,v\;\ell1{\mapsto}\{(1,1,\mathcal{L}(\ell1),1),(1,\epsilon,\mathcal{L}(\ell2),2),(2,\epsilon,\mathcal{L}(\ell1),3),(2,\epsilon,\mathcal{A},4)\},$$
$$x\;\ell2{\mapsto}\{(1,1,\mathcal{L}(\ell2),2)\},\;y\;\ell3{\mapsto}\{(2,1,\mathcal{L}(\ell1),3),(2,1,\mathcal{A},4)\}]{\subseteq}am$$

Isolation.

Isolated?(am) = True. Due to **C4**: the lock issued write and lock issued read by thread 1 and respectively thread 2 are isolated as they both use the same mutex, $\ell1$; transactional instance 4 accesses the mutex used by thread 1's lock

issued write of $\ell 1$, therefore are isolated; lock instances 2 and 3 do not need to use the same mutex as both only read $\ell 1$; likewise, transactional instance 4 does not need to access lock instance 2's mutex as both only read $\ell 1$. □

Example B.10 (Lock and Transactional Accesses)**.**

Program.

```
Int v; Int x; Int y;
```

```
v := 0; x := 0; y := 0;
```

Thread 1	**Thread 2**	**Thread 3**
`1:sync(v) {`	`3:sync(v) {`	`5:sync(x) {`
`v := 1;`	`y := v;`	`v := x;`
`}`	`}`	`}`
`2:atomic {`	`4:atomic {`	
`v := y;`	`y := v;`	
`}`	`}`	

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL) × 3, (ASSIGN−INT−LITERAL) × 3;
  - Thread 1: (LOCK)⟨(ASSIGN−INT−LITERAL)⟩, (TRANSACTION)⟨(ASSIGN−VAR−LITERAL)⟩;
  - Thread 2: (LOCK)⟨(ASSIGN−VAR−LITERAL)⟩, (TRANSACTION)⟨(ASSIGN−VAR−LITERAL)⟩;
  - Thread 3: (LOCK)⟨(ASSIGN−VAR−LITERAL)⟩;
- ⟩

Access Mapping.

$$[\,v\,\ell 1{\mapsto}\{(1,1,\mathcal{L}(\ell 1),1),(1,1,\mathcal{A},2),(2,\epsilon,\mathcal{L}(\ell 1),3),(2,\epsilon,\mathcal{A},4),(3,1,\mathcal{L}(\ell 2),5)\},$$
$$x\,\ell 2{\mapsto}\{(3,\epsilon,\mathcal{L}(\ell 2),5)\},\ y\,\ell 3{\mapsto}\{(1,\epsilon,\mathcal{A},2),(2,1,\mathcal{L}(\ell 1),3),(2,1,\mathcal{A},4)\}]{\subseteq}am$$

Isolation.

$\mathsf{Isolated?}(am) = \mathsf{False}$, due to **C4**. Thread 3's lock-issued write of $\ell 1$ is not isolated w.r.t. to the accesses issued to $\ell 1$ by threads 1 and 2. □

Example B.11 (Lock and Transactional Accesses)**.**

Program.

```
Int v; Int x; Int y;
v := 0; x := 0; y := 0;
```

Thread 1	**Thread 2**	**Thread 3**
`1:sync(v) {`	`3:sync(v) {`	`5:sync(v) {`
`v := 1;`	`y := v;`	`v := x;`
`}`	`}`	`}`
`2:atomic {`	`4:atomic {`	
`v := y;`	`y := v;`	
`}`	`}`	

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL) × 3, (ASSIGN−INT−LITERAL) × 3;
  - Thread 1: (LOCK)⟨(ASSIGN−INT−LITERAL)⟩,
    (TRANSACTION)⟨(ASSIGN−VAR−LITERAL)⟩;

– Thread 2: (LOCK)⟨(ASSIGN−VAR−LITERAL)⟩,
(TRANSACTION)⟨(ASSIGN−VAR−LITERAL)⟩;

– Thread 3: (LOCK)⟨(ASSIGN−VAR−LITERAL)⟩;

- ⟩

Access Mapping.

$$[\,v\,\ell 1 \mapsto \{(1,1,\mathcal{L}(\ell 1),1),(1,1,\mathcal{A},2),(2,\epsilon,\mathcal{L}(\ell 1),3),(2,\epsilon,\mathcal{A},4),(3,1,\mathcal{L}(\ell 1),5)\},$$
$$x\,\ell 2 \mapsto \{(3,\epsilon,\mathcal{L}(\ell 1),5)\},\; y\,\ell 3 \mapsto \{(1,\epsilon,\mathcal{A},2),(2,1,\mathcal{L}(\ell 1),3),(2,1,\mathcal{A},4)\}] \subseteq am$$

Isolation.

Isolated?(am) = True. $\ell 1$ is isolated due to **C4**, $\ell 2$ due to **C1** and $\ell 3$ due to **C4**. □

Example B.12 (Concurrently Mutating a Linked List)**.**

Program.

```
LinkedList l;
l := new LinkedList;
l.add(1)@nodefer;
l.add(2)@nodefer;
```

Thread 1	Thread 2
`l.add(3);`	`l.traverse();`
	`l.add(4);`

Rule Applications.

- (PROGRAM)⟨

- Main thread: (VAR−DECL), (NEW),
  (METHOD−CALL−ARG−NO−DEFER)⟨†⟩ × 2;
- Thread 1: (METHOD−CALL−DEFER);
- Thread 2: (METHOD−CALL−DEFER) × 2;
- Serialised Method Calls: (METHOD−CALL−ARG−DEFERRED)⟨†⟩ × 2,
  (METHOD−CALL−NO−ARG−DEFERRED)⟨∗⟩.

• ⟩

Where, † = (VAR−DECL), (NEW), (FLD−UPDATE−VAR−LITERAL), (FLD−UPDATE−FLD−REF), (FLD−UPDATE−VAR−REF); ∗ = (VAR−DECL), (ASSIGN−FLD−REF), (WHILE)⟨(NEQ),(PRINT),(ASSIGN−FLD−REF)⟩.

Variable and Entity Mappings. Application of the rules results in the following *var* and *obj* mappings, where *var* is an instance of Var and *obj* an instance of Obj. The structure of *var* and *obj* is diagrammatically shown in Figure B.1.

$$[l \mapsto (\ell 1, \ell 2)] \subseteq var$$

$$\begin{array}{lcl}
[\ell 2 & \mapsto & [\texttt{head} \mapsto (\ell 2, \ell 21)], \\
\ell 6 & \mapsto & [\texttt{next} \mapsto (\ell 6, \mathsf{null}), \texttt{value} \mapsto (\ell 7, \mathsf{null})], \\
\ell 11 & \mapsto & [\texttt{next} \mapsto (\ell 11, \ell 6), \texttt{value} \mapsto (\ell 12, \mathsf{null})], \\
\ell 16 & \mapsto & [\texttt{next} \mapsto (\ell 16, \ell 11), \texttt{value} \mapsto (\ell 17, \mathsf{null})], \\
\ell 21 & \mapsto & [\texttt{next} \mapsto (\ell 21, \ell 16), \texttt{value} \mapsto (\ell 22, \mathsf{null})]] \subseteq obj
\end{array}$$

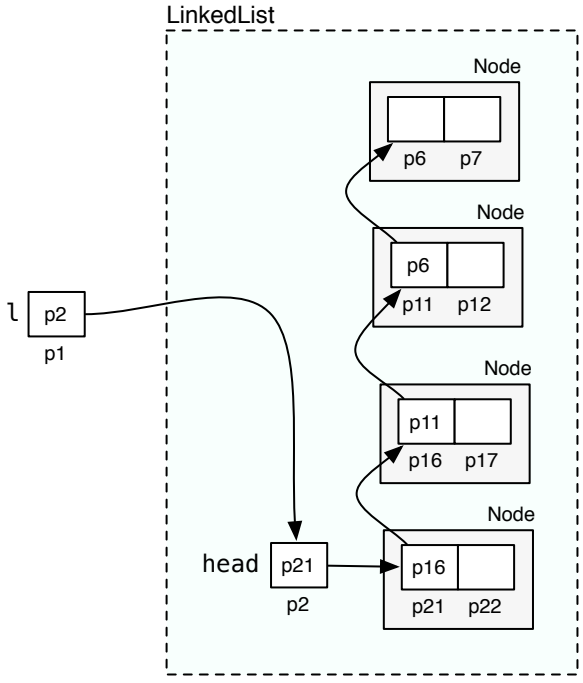


Figure B.1: Structure of the *anonymous* `LinkedList` object. The `LinkedList` object is anonymous due to all literal values being discarded – only the shape of the `LinkedList` that `l` points-to is of relevance.

Access Mapping.

We have omitted the memory locations associated with a method's formal parameters and locally defined variables as they do not escape.

$[$

l	(1)	$\ell 1 \mapsto \{(1, \epsilon, \bot, \bot), (2, \epsilon, \bot, \bot\}$,
LinkedList($head$)	(2)	$\ell 2 \mapsto \{(1, 1, \bot, \bot), (2, 1, \bot, \bot)\}$,
	(3_m)	`l.add(1)`@nodefer: `this`($\ell 3$), `val`($\ell 4$) and `n`($\ell 5$)
Node($next$	(3)	$\ell 6 \mapsto \{(2, \epsilon, \bot, \bot)\}$,
$value$)	(3)	$\ell 7 \mapsto \{(2, \epsilon, \bot, \bot)\}$,
	(4_m)	`l.add(2)`@nodefer: `this`($\ell 8$), `val`($\ell 9$) and `n`($\ell 10$)
Node($next$	(4)	$\ell 11 \mapsto \{(2, \epsilon, \bot, \bot)\}$,
$value$)	(4)	$\ell 12 \mapsto \{(2, \epsilon, \bot, \bot)\}$,
	(5_m)	`l.add(3)`@ctxt: `this`($\ell 13$), `val`($\ell 14$) and `n`($\ell 15$)
Node($next$	(5)	$\ell 16 \mapsto \{(1, 1, \bot, \bot), (2, \epsilon, \bot, \bot)\}$,
$value$)	(5)	$\ell 17 \mapsto \{(1, 1, \bot, \bot), (2, \epsilon, \bot, \bot)\}$,
	(6_m)	`l.add(4)`@ctxt: `this`($\ell 18$), `val`($\ell 19$) and `n`($\ell 20$)
Node($next$	(6)	$\ell 21 \mapsto \{(2, 1, \bot, \bot)\}$,
$value$)	(6)	$\ell 22 \mapsto \{(2, 1, \bot, \bot)\}$,
	(7_m)	`l.traverse()`@ctxt: `this`($\ell 23$)and `curr`($\ell 24$)

$]\subseteq am$

The domain of am reveals our example program allocated 24 memory locations during its static execution. The labels $(1) \ldots (7)$ correspond to the following descriptions:

1. `LinkedList l` variable declared by the main thread;
2. `LinkedList` instance allocated by the main thread;
3. Node instance allocated by the main thread's invocation of `l.add(1)`@nodefer. The omitted memory locations $\ell 3$, $\ell 4$ and $\ell 5$ in (3_m) were allocated to support the invocation of `add`;
4. `l.add(2)`@nodefer invoked by the main thread;
5. `l.add(3)`@ctxt invoked by thread 1;
6. `l.add(4)`@ctxt invoked by thread 2;
7. `l.traverse()`@ctxt invoked by thread 2.

Isolation.

Isolated?(am) = False, due to **C2.3**. Thread 1's write of $\ell 2$, `l.head` in `l.add(3)`, is not isolated with respect to thread 2's accesses of $\ell 2$ in `l.add(4)` and `l.traverse()`. □

Example B.13 (Concurrently Mutating a Linked List using Transactions)**.** We will attempt to give an isolated version of the program given in Example B.12 by using transactions.

Program.

```
LinkedList l;
l := new LinkedList;
l.add(1)@nodefer;
```

`l.add(2)@nodefer;`

Thread 1	Thread 2
`1:atomic {`	`2:atomic {`
`l.add(3);`	`l.traverse();`
`}`	`l.add(4);`
	`}`

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL), (NEW), (METHOD−CALL−ARG−NO−DEFER)⟨†⟩ × 2;
  - Thread 1: (TRANSACTION)⟨(METHOD−CALL−DEFER)⟩;
  - Thread 2: (TRANSACTION)⟨(METHOD−CALL−DEFER) × 2⟩;
  - Serialised Method Calls: (METHOD−CALL−ARG−DEFERRED)⟨†⟩ × 2, (METHOD−CALL−NO−ARG−DEFERRED)⟨∗⟩.
- ⟩

Where, † = (VAR−DECL), (NEW), (FLD−UPDATE−VAR−LITERAL), (FLD−UPDATE−FLD−REF), (FLD−UPDATE−VAR−REF); ∗ = (VAR−DECL), (ASSIGN−FLD−REF), (WHILE)⟨(NEQ),(PRINT) ⇝ ⊥.

Unfortunately, application of our rules does not complete due to (PRINT) yielding an undefined environment. This occurs due to CheckSafeIO failing in the premise of (PRINT). Consequently, the program is pessimistically declared not isolated. □

Example B.14 (Concurrently Mutating a Linked List using Transactions and Locks)**.** We now modify the program given in Example B.13 to execute thread 2's commands within a lock to address the weak execution semantics of transactions.

Program.

```
LinkedList l;
l := new LinkedList;
l.add(1)@nodefer;
l.add(2)@nodefer;
```

Thread 1	**Thread 2**
`1:atomic {`	`2:sync(l) {`
`l.add(3);`	`l.traverse();`
`}`	`l.add(4);`
	`}`

Rule Applications.

- (PROGRAM)⟨
  - Main thread: (VAR−DECL), (NEW), (METHOD−CALL−ARG−NO−DEFER)⟨†⟩ × 2;
  - Thread 1: (TRANSACTION)⟨(METHOD−CALL−DEFER)⟩;
  - Thread 2: (LOCK)⟨(METHOD−CALL−DEFER) × 2⟩;
  - Serialised Method Calls: (METHOD−CALL−ARG−DEFERRED)⟨†⟩ × 2, (METHOD−CALL−NO−ARG−DEFERRED)⟨∗⟩.
- ⟩

Where, † = (VAR−DECL), (NEW), (FLD−UPDATE−VAR−LITERAL), (FLD−UPDATE−FLD−REF), (FLD−UPDATE−VAR−REF); ∗ = (VAR−DECL), (ASSIGN−FLD−REF), (WHILE)⟨(NEQ),(PRINT),(ASSIGN−FLD−REF)⟩.

Access Mapping.

[

l	(1)	$\ell 1 \mapsto \{(1, \epsilon, \mathcal{A}, 1), (2, \epsilon, \mathcal{L}(\ell 1), 2)\}$,
LinkedList($head$)	(2)	$\ell 2 \mapsto \{(1, 1, \mathcal{A}, 1), (2, 1, \mathcal{L}(\ell 1), 2)\}$,
	(3_m)	`l.add(1)`@nodefer: `this`$(\ell 3)$, `val`$(\ell 4)$ and `n`$(\ell 5)$
Node($next$	(3)	$\ell 6 \mapsto \{(2, \epsilon, \mathcal{L}(\ell 1), 2)\}$,
$value$)	(3)	$\ell 7 \mapsto \{(2, \epsilon, \mathcal{L}(\ell 1), 2)\}$,
	(4_m)	`l.add(2)`@nodefer: `this`$(\ell 8)$, `val`$(\ell 9)$ and `n`$(\ell 10)$
Node($next$	(4)	$\ell 11 \mapsto \{(2, \epsilon, \mathcal{L}(\ell 1), 2)\}$,
$value$)	(4)	$\ell 12 \mapsto \{(2, \epsilon, \mathcal{L}(\ell 1), 2)\}$,
	(5_m)	`l.add(3)`@ctxt: `this`$(\ell 13)$, `val`$(\ell 14)$ and `n`$(\ell 15)$
Node($next$	(5)	$\ell 16 \mapsto \{(1, 1, \mathcal{A}, 1), (2, \epsilon, \mathcal{L}(\ell 1), 2)\}$,
$value$)	(5)	$\ell 17 \mapsto \{(1, 1, \mathcal{A}, 1), (2, \epsilon, \mathcal{L}(\ell 1), 2)\}$,
	(6_m)	`l.add(4)`@ctxt: `this`$(\ell 18)$, `val`$(\ell 19)$ and `n`$(\ell 20)$
Node($next$	(6)	$\ell 21 \mapsto \{(2, 1, \mathcal{L}(\ell 1), 2)\}$,
$value$)	(6)	$\ell 22 \mapsto \{(2, 1, \mathcal{L}(\ell 1), 2)\}$,
	(7_m)	`l.traverse()`@ctxt: `this`$(\ell 23)$and `curr`$(\ell 24)$

$]\subseteq am$

Isolation.

Isolated?(am) = True. A total ordering exists over the accesses performed by thread 1 and 2's transaction and lock. There are two points of contention:

- $p2$ – Thread 1's invocation of `add` needs to be isolated with thread 2's invocations of `add` and `traverse` because each invocation of `add` writes `l.head` at memory location $\ell 2$, and `traverse` reads $\ell 2$. Each thread's accesses are isolated as transactional instance 1 accesses the mutex $\ell 1$ used by thread 2's lock which protects its invocations of `add` and `traverse`.

- $\ell 16$ and $\ell 17$ – The `Node` allocated by thread 1's invocation of `add` needs to be isolated with respect to thread 2's invocation of traverse due to the allocated node being reachable by `traverse`. The invocation of `add` by transactional instance 1 is isolated with respect to thread 2's lock issued `traverse` due to transactional instance 1 accessing the mutex $\ell 2$ which protects the invocation of `traverse`.

The second point of contention is a problem due to the possible semantics of the underlying memory model. For example, in the schedule `l.add(); || l.traverse(); traverse` may not observe the state of thread 1's allocated `Node` due to the accesses issued by each method not being related by the underlying memory model. That is, the writes issued by thread 1's invocation of `add` may be buffered and not flushed to main memory before the reads issued by `traverse` take place. In the Java memory model Manson et al. [2005] we might say, assuming a transaction has appropriately defined synchronisation actions and relationships within synchronises-with, that the accesses issued by thread 2's invocation of `traverse` and thread 1's `add` are not related in happens-before. Therefore, a data race may occur. □

References

Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. Principles and Practice of Parallel Programming. ACM, 2009.

Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer, IEEE Transactions on*, 29, 1996.

Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 1st edition, 2010.

Joe Armstrong, Robert Virding, Claes Wikstr, Mike Williams, et al. *Concurrent programming in Erlang*. Prentice Hall, 1st edition, 1996.

Ken Arnold, James Gosling, and David Holmes. *The Java(TM) Programming Language*. Addison-Wesley Professional, 4th edition, 2005.

Granville Barnett and Shengchao Qin. Moverness for locks and transactions. Theoretical Aspects of Software Engineering. IEEE, 2012a.

Granville Barnett and Shengchao Qin. A composable mixed mode concurrency control semantics for transactional programs. International Conference on Formal Engineering Methods. Springer-Verlag, 2012b.

Granville Barnett and Shengchao Qin. Data-race-freedom of concurrent programs. Asia-Pacific Software Engineering Conference. IEEE, 2013.

Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. Object-Oriented Programming, Systems, Languages and Applications. ACM, 2008.

Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control theory and algorithms. *Database Systems, ACM Transactions on*, 1983.

Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356–368. IEEE, 1994.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. Principles and Practice of Parallel Computing. ACM, 1995.

Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. PLDI. ACM, 2008.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. Principles of Programming Languages. ACM, 2005.

Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly Media, 3rd edition, 2005.

John Boyland. Checking interference with fractional permissions. Static Analysis Symposium. Springer-Verlag, 2003.

John Tang Boyland. Semantics of fractional permissions with nesting. *TOPLAS*, 2010.

David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1997.

Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 2009.

Chromium-Project. Chromium browser, 2013. URL http://www.chromium.org/Home.

Ariel Cohen. *Verification of Transactional Memories and Recursive Programs*. PhD thesis, Department of Computer Science, New York University, 2008.

Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. Distributed Computing. IEEE, 2006.

E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 1983.

Edsger W. Dijkstra. The structure of the "the" multiprogramming system. *Communications of the ACM*, 1968.

Jeremie Dimino. Lwt user manual, July 2012. URL http://ocsigen.org/lwt/files/manual.pdf.

Joe Duffy. *Concurrent Programming on Windows*. Addison-Wesley Professional, 1st edition, 2008.

Peyton Jones (editor), Simon, John Hughes (editor), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from http://www.haskell.org/definition/, February 1999.

Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. Haskell. ACM, 2011.

Rob Farber. *CUDA Application Design and Development*. Morgan Kaufmann Publishers Inc., 1st edition, 2011.

David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly Media, 1st edition, 2008.

Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2), 2007.

Rakash Ghiya and Laurie Hendren. Is it a tree, a dag, or a cyclic graph? Principles of Programming Languages. ACM, 1996.

Google-Go. The go programming language, April 2013. URL http://golang.org/.

Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? Memory System Performance and Correctness. ACM, 2006.

Rachid Guerraoui and Micha Kapaka. Opacity: A correctness condition for transactional memory. Technical report, EPFL, 2007.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. Principles and Practice of Parallel Programming. ACM, 2005.

Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# Programming Language (Covering C# 4.0)*. Addison-Wesley Professional, 2010.

Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. Principles and Practice of Parallel Programming. ACM, 2008.

Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. International Symposium on Computer Architecture. ACM, 1993.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 1st edition, 2008.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. Principles of Distributed Computing. ACM, 2003.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *Programming Languages and Systems, ACM Transactions on*, 1990.

Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Fractional permissions without the fractions. Formal Techniques for Java-like Programs. ACM, 2011.

Rich Hickey. The clojure programming language. Dynamic Languages Symposium. ACM, 2008.

C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 1974.

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

Liyang Hu. *Compiling Concurrency Correctly Verifying Software Transactional Memory*. PhD thesis, University of Nottingham, 2012.

Intel. Intel c++ stm compiler, prototype edition, January 2012. URL http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition.

Intel. Hyper-threading, March 2013a. URL www.intel.com/info/hyperthreading.

Intel. Intel cilk plus, April 2013b. URL http://software.intel.com/en-us/intel-cilk-plus.

ISO-WG21. Transactional memory, November 2012. URL http://isocpp.org/std/the-committee.

James Jenista and Brian Demsky. Disjointness analysis for java-like languages. Technical report, University of California, Irvine, 2009.

Richard Jones and Rafael D Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, 1st edition, 1996.

Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

Nicolai Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison Wesley, 2nd edition, 2012.

Michael Kerrisk. *The Linux programming interface*. No Starch Press, 1st edition, 2010.

Stephen Kochan. *Programming in Objective-C*. Sams, 5th edition, 2012.

Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. Coarse-grained transactions. Principles of Programming Languages. ACM, 2010.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Transactions on Computers, IEEE Transactions on*, 1979.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

Douglas Lea. *Concurrent Programming in Java(TM): Design Principles and Patterns*. Addison-Wesley Professional, 3rd edition, 2006.

K.RustanM. Leino, Peter Mller, and Jan Smans. Verification of concurrent programs with chalice. Lecture Notes in Computer Science. Springer-Verlag, 2009.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.00. 2012.

Y. Lev and J.-W. Maessen. Towards a safer interaction with transactional memory by tracking object visibility. SCOOL. ACM, 2005.

Yossi Lev, Victor Luchangco, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. Transactional Computing. ACM, 2009.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Addison Wesley, java se 7 edition, 2013.

D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM SIGSOFT Software Engineering Notes*, 1977.

Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. Principles of Programming Languages. ACM, 2005.

Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. Principles of Programming Languages. ACM, 2006.

Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity

semantics for java stm. Symposium on Parallelism in Algorithms and Architectures. ACM, 2008.

Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, 3rd edition, 2005.

Microsoft. Sql server replication, 2012. URL http://msdn.microsoft.com/en-us/library/ms151198.aspx.

Microsoft. C++ amp overview, April 2013a. URL http://msdn.microsoft.com/en-us/library/vstudio/hh265136.aspx.

Microsoft. Microsoft concurrency runtime, April 2013b. URL http://msdn.microsoft.com/en-us/library/vstudio/dd504870.aspx.

Microsoft. Microsoft task parallel library, April 2013c. URL http://msdn.microsoft.com/en-gb/library/dd460717.aspx.

Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. High-Performance Computer Architecture. IEEE, 2006.

Mozilla-Rust. Rust, April 2013. URL http://www.rust-lang.org/.

Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. Programming Language Design and Implementation. ACM, 2007.

Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design

and implementation of transactional constructs for c/c++. Object-Oriented Programming Systems Languages and Applications. ACM, 2008.

Scott Oaks and Henry Wong. *Java Threads*. O'Reilly Media, Inc., 3rd edition, 2004.

Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Incorporation, 2nd edition, 2011.

Chris Okasaki. *Purely functional data structures*. PhD thesis, Carnegie Mellon University, 1996.

Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. Architectural Support for Programming Languages and Operating Systems. ACM, 1996.

Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 1st edition, 2005.

James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 1st edition, 2010.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. LICS. IEEE, 2002.

Jeffrey Richter. *CLR via C#*. Microsoft Press, 4th edition, 2012.

Dennis M Ritchie and Kernighan. *The C programming language*. Prentice Hall, 2nd edition, 1988.

Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows® Internals*. Microsoft Press, 6th edition, 2012.

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. Principles and Practice of Parallel Programming. ACM, 2006.

Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 1st edition, 2010.

Robert R Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34 (6):52–59, 1997.

William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. Principles of Distributed Computing. ACM, 2005.

Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. Workshop on Binary Instrumentation and Applications. ACM, 2009.

Nir Shavit and Alex Matveev. Towards a fully pessimistic stm model. Transactional Computing. ACM, 2012.

Nir Shavit and Dan Touitou. Software transactional memory. Principles of Distributed Computing. ACM, 1995.

Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. Object-Oriented Programming Systems Languages and Applications. ACM, 2007.

Nehir Sonmez, Tim Harris, Adrian Cristal, Osman S. Unsal, and Mateo Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. International Parallel and Distributed Processing Symposium. IEEE, 2009.

Michael F. Spear, Virendra J. Marathe, Luke Daless, and Michael L. Scott. Privatization techniques for software transactional memory. Principles of Distributed Computing. ACM, 2007.

Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. Ordering-based semantics for software transactional memory. On Principles of Distributed Systems. Springer-Verlag, 2008.

Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. Principles and Practice of Parallel Programming, 2009.

Alexander Stepanov and Meng Lee. *The standard template library.* Hewlett-Packard Laboratories, Technical Publications Department, 1995.

Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2000.

David Stutz, Tim Neward, and Geoff Shilling. *Shared Source CLI Essentials.* O'Reilly Media, 1st edition, 2003.

Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 2005.

TypeSafe. Akka, April 2013. URL http://akka.io/.

Stephen H. Unger. Hazards, critical races, and metastability. IEEE Transactions on Computers, 1995.

Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. Parallel Architectures and Compilation Techniques. IEEE, 2009.

Valgrind-Project. Helgrind: a thread error detector, 2013. URL http://valgrind.org/docs/manual/hg-manual.html.

Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. Symposium on Parallelism in Algorithms and Architectures. ACM, 2008.

Anthony Williams. *C++ concurrency in action*. Manning, 1st edition, 2012.

Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for java. European Conference on Object-Oriented Programming. Springer-Verlag, 2008.

Dieter Zöbel. The deadlock problem: a classifying bibliography. *ACM SIGOPS Operating Systems Review*, 1983.